# BIOSEQ: UNA LIBRERÍA PARA BIOINFORMÁTICA EN R

JORGE MARTÍNEZ LADRÓN DE GUEVARA

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA, FACULTAD DE INFORMÁTICA,
UNIVERSIDAD COMPLUTENSE DE MADRID


Trabajo Fin Máster en Ingeniería de Computadores

Junio, 2013

Directora:
Victoria López

Colaboradora de dirección:
Beatriz González

# Autorización de Difusión

JORGE MARTÍNEZ LADRÓN DE GUEVARA

Junio, 2013

El abajo firmante, matriculado en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: "BIOSEQ: UNA LIBRERÍA PARA BIOINFORMÁTICA EN R", realizado durante el curso académico 2012-2013 bajo la dirección de Victoria López y la colaboración de dirección de Beatriz González en el Departamento de Arquitectura de Computadores, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo en Internet y garantizar su preservación y acceso a largo plazo.

# Resumen en castellano

Este trabajo desarrolla temas de programación avanzada en R, como la programación orientada a objetos o el desarrollo de librerías externas C para el diseño y desarrollo de paquetes R. Además, describe la estructura de carpetas y archivos de un paquete R y el proceso de verificación y compilación necesario para su distribución. Por último, describe las funciones Smith-Waterman y Needleman-Wunsch para alineamiento de secuencias, incluidas en la librería BioSeq.

# Palabras clave

Programación en R, clases S3, clases S4, librerías externas C para R, desarrollo de paquetes en R, análisis de secuencias, Smith-Waterman, Needleman-Wunsch, bioinformática

# Resumen en inglés

This document focuses on advanced topics in R programming useful for package development, covering Object Oriented Programming and the interface with C libraries. This work also describes the structure of an R packages and its contents, and the compilation process necessary for distribution. Finally, it describes Smith-Waterman and Needleman-Wunsch functions for sequence alignment which are included in BioSeq package.

# Keywords

R programming language, S3 classes, S4 classes, C external libraries for R, package development in R, sequence analysis, Smith-Waterman, Needleman-Wunsch, bioinformatics

# Índice de contenidos

Autorización de Difusión ..... i
Resumen en castellano ..... iii
Palabras clave ..... iii
Resumen en inglés ..... v
Keywords ..... v
Índice de contenidos ..... vii
Capítulo 1. Introducción ..... 1
Capítulo 2. El lenguaje de programación R ..... 5
2.1. Fundamentos ..... 6
2.1.1. Operadores y funciones ..... 6
2.1.2. Expresiones ..... 7
2.1.3. El espacio de trabajo y los objetos ..... 8
2.2. Tipos de datos ..... 8
2.2.1. Numeric ..... 9
2.2.2. Logical ..... 9
2.2.3. Character ..... 10
2.2.4. Factor ..... 13
2.2.5. Date ..... 13
2.2.6. NA, NaN, NULL ..... 15
2.2.7. Conversión de tipos de datos ..... 15
2.2.8. La función typeof() ..... 16
2.3. Estructuras de datos ..... 17
2.3.1. Vector ..... 17
2.3.2. Matriz ..... 22
2.3.3. Array ..... 26
2.3.4. Lista ..... 27
2.3.5. Dataframe ..... 30
2.4. Estructuras de selección ..... 38
2.4.1. if ..... 38
2.4.2. if else ..... 38
2.4.3. switch ..... 40
2.5. Estructuras de repetición ..... 40
2.5.1. while ..... 41
2.5.2. repeat ..... 41
2.5.3. for ..... 42

2.6. Funciones ..... 43
2.6.1. Declaración de funciones ..... 43
2.6.2. Argumentos y valores por defecto ..... 44
2.6.3. El argumento '…' ..... 45
2.6.4. Evaluación de funciones ..... 46
2.7. Ámbito léxico ..... 46
2.8. Excepciones ..... 47
2.8.1. Funciones de manejo de excepciones ..... 47
2.8.2. Opciones de control de errores ..... 47
2.9. Importación y exportación de datos ..... 48
2.9.1. La función read.table() ..... 48
2.9.2. La función scan() ..... 50
2.9.3. La función read.csv() ..... 51
2.9.4. La función write.csv() ..... 51
2.9.5. La función sink() ..... 51
2.9.6. La función capture.output() ..... 52
2.10. Gráficos ..... 52
Capítulo 3. Programación orientada a objetos ..... 55
3.1. Clases S3 ..... 56
3.1.1. Declaración de clases ..... 56
3.1.2. Declaración de métodos ..... 57
3.1.3. Funciones de uso común ..... 62
3.2. Clases S4 ..... 62
3.2.1. Declaración de clases ..... 62
3.2.2. Declaración de métodos ..... 65
3.2.3. Extensión de clases ..... 70
3.2.4. Clases virtuales ..... 75
3.2.5. Funciones de uso común ..... 78
Capítulo 4. Desarrollo de librerías en C para R ..... 79
4.1. La función .C ..... 79
4.2. El código C ..... 80
4.3. El código R ..... 81
4.4. Uso de funciones externas C y eficiencia ..... 81
4.5. Conversión de tipos de datos entre R y C ..... 86
Capítulo 5. Recomendaciones de estilo para R ..... 89
5.1. Identificadores y nombres ..... 89
5.1.1. Archivos ..... 90
5.1.2. Variables ..... 90
5.1.3. Funciones ..... 90
5.1.4. Clases ..... 91

5.2. Sintaxis ..... 91
5.2.1. Longitud de las líneas de código ..... 91
5.2.2. Uso de los espacios en blanco ..... 91
5.2.3. Sangría del código ..... 92
5.2.4. Uso de llaves ..... 92
5.2.5. El operador de asignación ..... 93
5.2.6. Uso de return(), invisible() ..... 93
5.3. Comentarios del código ..... 93
5.4. Objetos y métodos ..... 93
5.5. Manejo de excepciones ..... 94
5.6. Documentación de funciones ..... 94
Capítulo 6. Desarrollo de paquetes en R ..... 95
6.1. Estructura de un paquete R ..... 95
6.2. El contenido de un paquete ..... 96
6.2.1. El fichero DESCRIPTION ..... 96
6.2.2. El fichero NAMESPACES ..... 97
6.2.3. Los ficheros de documentación ..... 97
6.2.4. Los ficheros de código R ..... 99
6.2.5. Los ficheros de datos ..... 99
6.3. El proceso de verificación y compilación de un paquete ..... 100
Capítulo 7. La librería BioSeq para bioinformática ..... 103
7.1. Funciones de alineamiento de secuencias ..... 104
7.1.1. SmithWaterman ..... 104
7.1.2. NeedlemanWunsch ..... 106
7.2. Bases de datos ..... 107
7.3. El código R ..... 109
Capítulo 8. Conclusiones ..... 115
Referencias bibliográficas ..... 117

# Capítulo 1. Introducción

R es un entorno integrado de aplicaciones informáticas diseñado para facilitar la manipulación de datos, realizar cálculos estadísticos y gráficos. Proporciona un lenguaje de programación, gráficos e 'interfaces' para C, C++ y Fortran. R se aplica en diversas áreas de conocimiento como la estadística, la investigación biomédica, la bioinformática o las matemáticas financieras. R es un proyecto de software libre resultado de implementar el lenguaje S bajo el sistema operativo GNU[1].

R es un lenguaje de programación y un entorno de ejecución de aplicaciones orientadas al análisis de datos. Las características más destacadas de R son:

- El entorno de R facilita la manipulación eficiente de datos y su almacenamiento. Ofrece un amplio conjunto de herramientas para análisis de datos, funciones gráficas y operadores para realizar cálculos con vectores y matrices
- Las aplicaciones R son portables y se pueden ejecutar en los principales sistemas operativos del mercado: Windows , Linux, Mac OS
- El desarrollo de paquetes permite la libre distribución de software y datos destinados a cubrir necesidades específicas de diversas especialidades. El uso de R está muy extendido y es utilizado en proyectos de investigación
- En el desarrollo de R colaboran científicos de alto nivel del ámbito de la estadística y la informática. R se distribuye bajo licencia GNU GPL[2]. El software es libre y de código abierto. R es gratuito y se dispone del código fuente del lenguaje y de los paquetes

R es un lenguaje extensible, los usuarios de R pueden publicar paquetes para extender su funcionalidad. En realidad, el lenguaje R es parte de un proyecto colaborativo y abierto, que dispone de un repositorio oficial de más de 2.000 paquetes. El sitio web CRAN[3], almacena todos los recursos software, los paquetes y la documentación técnica de R.

El desarrollo de paquetes en R es complejo. Dependiendo de los requisitos del paquete, puede requerir conocimientos avanzados de programación en R y en C, sobre todo

---

[1] GNU es un acrónimo de "Gnu No es Unix". En 1984 comenzó el desarrollo de un sistema operativo completo como software libre.
[2] GNU GPL (GNU General Public License) es la licencia pública general de GNU.
[3] The Comprehensive R Archive Network (CRAN) es el sitio web donde se almacenan todos los recursos de software y la documentación del lenguaje R.

cuando se exige que el paquete sea robusto y eficiente. Para diseñar y desarrollar un paquete de calidad, se debe aplicar una metodología de desarrollo de software y realizar un estudio previo para saber si la funcionalidad que se quiere desarrollar ya existe en otros paquetes. De esta manera, se puede identificar claramente cuál es el valor que aporta el nuevo paquete. Una vez realizado el estudio previo se debe analizar y diseñar el conjunto de funciones que van a definir la interfaz del paquete. ¿Qué funciones van a estar disponibles para el usuario? ¿cómo se usan? ¿para qué sirven? ¿cuáles son sus argumentos? ¿cuál es su valor de retorno? ¿qué estructura de clases se va a utilizar? ¿qué métodos es necesario desarrollar para los objetos de estas clases? También se debe definir desde el principio si el paquete va a almacenar bases de datos. ¿Qué bases de datos va a incluir el paquete? ¿cuál es su estructura? ¿qué campos tienen? ¿qué tipos de datos va a almacenar cada campo? Es importante realizar el diseño de las funciones y las bases de datos del paquete en la fase inicial del proyecto y evitar modificar este diseño, ya que esto afecta a las fases de desarrollo, pruebas y documentación de todo el paquete.

El desarrollo de paquetes a menudo requiere conocimientos de programación orientada a objetos utilizando clases S3 o S4. El uso de clases permite estructurar y encapsular el código de la aplicación y facilita su reutilización. Si las funciones del paquete se desarrollan aplicando estas técnicas de programación durante la fase de construcción del paquete, esto facilita las tareas de mantenimiento y la incorporación de nuevas funcionalidades en el futuro. Si además se exige que las funciones del paquete tengan un buen rendimiento con grandes cantidades de datos, entonces es imprescindible optimizar las funciones desarrollando librerías externas en C.

Una de las dificultades que he encontrado durante la realización de este trabajo ha sido la necesidad de utilizar diversas fuentes bibliográficas para entender de forma clara temas importantes como la programación orientada a objetos, el desarrollo de librerías en C o las aspectos clave del proceso de verificación y compilación de paquetes. Me ha llamado la atención que el documento 'R Language Definition', una de las referencias básicas del lenguaje apenas desarrolla el sistema de clases S3 e incluso tiene apartados que están incompletos[4]. Evidentemente, la propia complejidad del lenguaje, sumada a la diversidad de la bibliografía y a la calidad de la documentación disponible, aumenta la dificultad de desarrollar aplicaciones R de calidad. Por estas razones, este trabajo abarca temas que van desde las estructuras básicas de datos de R hasta temas avanzados como la programación orientada a objetos y el desarrollo de librerías en C. El objetivo es que

[4] La sección 5.2 de este documento incluye el comentario 'FIXME Somethig is missing here'

sea útil y sirva como guía para desarrollar paquetes R a personas que tengan un nivel medio de conocimientos de programación de este lenguaje.

Este documento se organiza en los siguientes apartados:

El capítulo 2 es una guía rápida de programación en R, incluye operadores, tipos de datos, estructuras de datos, estructuras de selección, estructuras de repetición, aspectos importantes sobre el desarrollo de funciones, manejo de excepciones, importación y exportación de datos y gráficos. Este apartado se enfoca en las características fundamentales del lenguaje, necesarias para desarrollar cualquier tipo de aplicación y no en su uso para análisis estadístico, como muchos otros manuales y guías de programación de R.

El capítulo 3 desarrolla la programación orientada a objetos con los sistemas de clases S3 y S4, dada su importancia para la reutilización del código y el mantenimiento de las aplicaciones. Además, el uso correcto de estas técnicas de programación mejora la calidad de las aplicaciones y evita errores durante el proceso de verificación y compilación de un paquete. Este apartado desarrolla la declaración de clases S3, la sobreescritura de métodos de R y la declaración de métodos propios de las clases S3. Asimismo, desarrolla la declaración de clases S4, los métodos de instanciación y validación de objetos, la sobreescritura de métodos de R, la declaración de métodos propios de las clases S4, la extensión de clases y las clases virtuales. Con la finalidad de mostrar las características y las limitaciones de la programación orientada a objetos de R, se desarrolla el mismo ejemplo aplicando los sistemas de clases S3 y S4 para facilitar su comparación.

El capítulo 4 describe los conceptos fundamentales del desarrollo de librerías externas en C y su integración con aplicaciones R. Este apartado incluye un análisis comparativo del rendimiento de tres funciones que calculan la covarianza entre dos vectores. La primera de ellas utiliza la función `cov`, nativa de R; la segunda función utiliza una librería externa C y la última está desarrollada utilizando estructuras de repetición de R. Los resultados son concluyentes y queda claro que es necesario desarrollar librerías externas C para diseñar aplicaciones R que sean eficientes con grandes cantidades de datos.

El capítulo 5 propone recomendaciones básicas de estilo de programación en R. Esto se justifica porque, a pesar de que R es un proyecto abierto y hay mucha gente colaborando en el desarrollo de paquetes, no existe una norma definida por el equipo responsable del lenguaje (R Development Core Team).

El capítulo 6 desarrolla el proceso de verificación y construcción de un paquete para Windows. Describe la estructura de un paquete y los distintos archivos que lo

componen. Además, se detalla el software necesario para compilar paquetes en Windows.

El capítulo 7 describe la librería `BioSeq` para alineamiento de secuencias. Este paquete ha sido desarrollado en colaboración con otros alumnos de Bioinformática del Máster en Investigación en Informática.

Por último, el capítulo 8 desarrolla las conclusiones y las líneas de trabajo futuro.

# Capítulo 2. El lenguaje de programación R

R es un lenguaje de programación interpretado, de ahí que los programas R son portables porque no se compilan para un sistema operativo en particular, sino que se ejecutan paso a paso por un programa que interpreta los comandos del lenguaje. R toma muchas características del lenguaje S, diseñado específicamente para el análisis estadístico. S fue desarrollado por John Chambers[5], Richard Becker y Allan Wilks entre 1975 y 1976 en el departamento de análisis estadístico de los laboratorios Bell. En esa época, los sistemas informáticos de análisis estadístico de los laboratorios Bell se realizaban utilizando la librería Fortran SCS (Statistical Computing Subroutines). El objetivo de S era desarrollar un lenguaje y un entorno de programación interactivo con capacidad para realizar gráficos. S fue diseñado como un lenguaje interactivo basado en funciones parametrizadas para realizar análisis estadístico y análisis de datos. Según John Chambers, el objetivo de S era claro: "convertir ideas en software, de forma rápida y fiable para facilitar el análisis estadístico". S es un lenguaje orientado a los datos y ofrece herramientas de uso general para organizar, almacenar y recuperar distintos tipos de datos. Además, S ofrece métodos numéricos y otras técnicas que facilitan el cálculo y la interpretación de los datos, así como 'interfaces' para comunicarse con el sistema operativo y con rutinas en C, C++ o Fortran [2].

En 1993, Ross Ihaka y Robert Gentleman crean R en la Univesidad de Auckland. En junio de 1995, Ihaka y Gentleman deciden distribuir R bajo la licencia general de la fundación GNU de software libre y abierto. En 1997 se forma el R Development Core Team y en 2000 sale la versión 1.0. Un año después, en 2001, se publica el primer número de R-News. Al año siguiente se crea The R Foundation for Statistical Computing y en 2009 el R-Journal sustituye a R-News. Actualmente el desarrollo de R es responsabilidad del grupo R Development Core Team [19, 29].

La sintaxis de R tiene cierto parecido con C, pero su semántica es la de un lenguaje de programación funcional e incorpora características de LISP, APL y AWK. R permite declarar funciones que utilizan expresiones como argumentos, lo que de gran utilidad cuando se trabaja con modelos estadísticos y gráficos. Además, facilita el desarrollo de funciones de cálculo y tratamiento de datos que se pueden almacenar en nuevos paquetes. Los paquetes de R almacenan funciones y datos, para que el contenido de un paquete esté disponible, es necesario cargarlo en el entorno de ejecución de R.

[5] En 1998, la Association for Computing Machinery (ACM) reconoció el trabajo de John Chambers en el sistema S, que ha definido una nueva forma en que los usuarios analizan, visualizan y manipulan datos.

Este capítulo describe las características más significativas de la programación en R, desde las estructuras de datos básicas del lenguaje hasta temas de programación avanzada necesarios para el desarrollo de paquetes y librerías [19, 22, 25, 27-29].

## 2.1. Fundamentos

R es un lenguaje interactivo. La consola de R permite realizar cálculos simples y cálculos avanzados utilizando las funciones propias del lenguaje y las funciones almacenadas en los paquetes del sistema.

### 2.1.1. Operadores y funciones

La consola de R ejecuta expresiones utilizando la línea de comandos del entorno de trabajo. El resultado de la expresión se muestra por pantalla a la vez que se almacena en el objeto `.Last.value`.

```
> 2+3*6                  # precedencia de operadores: ^,*,/,+,-
[1] 20
> (2+3)*6                # paréntesis y precedencia de operadores
[1] 30
> 3/2                    # división
[1] 1.5
> 3 %/% 2                # división entera
[1] 1
> 2^2^2^2                # potencia
[1] 65536
> sqrt(25)               # raíz cuadrada
[1] 5
> abs(3-5)               # la función valor absoluto
[1] 2
> pi                     # el número pi
[1] 3.141593
> log(10)                # logaritmo natural
[1] 2.302585
> exp(1)                 # la función exponencial
[1] 2.718282
```

El operador de asignación '`<-`' almacena el valor del lado derecho de la expresión en el objeto indicado a la izquierda del operador. R es sensible a mayúsculas y minúsculas, por lo que `x` y `X` son dos objetos distintos.

```
> z <- 2*pi
> z
[1] 6.283185
> floor(z)               # mayor entero menor o igual a z
[1] 6
> ceiling(z)             # menor entero mayor o igual a z
[1] 7
```

Los números complejos en R.

```
> x <- -4+2i             # declaración del número complejo x
> x
[1] -4+2i
```

```
> Re(x)                    # parte real de x
[1] -4
> Im(x)                    # parte imaginaria de x
[1] 2
> y <- -1+1i               # declaración del número complejo y
> y
[1] -1+1i
> x+y                      # suma de dos números complejos
[1] -5+3i
> x*y                      # producto de dos números complejos
[1] 2-6i
> x/y                      # división de dos números complejos
[1] 3+1i
```

### 2.1.2. Expresiones

Una expresión contiene una o más sentencias del lenguaje R. El cálculo de cualquier expresión de R consiste de la evaluación secuencial de las sentencias de las que se compone. Una sentencia se separa de otra con símbolo '`;`' o con un salto de línea. Las sentencias se pueden agrupar utilizando los símbolos '`{`' y '`}`'. Las expresiones aritméticas de R admiten el uso de operadores aritméticos de forma similar a como se hace en una expresión del lenguaje C.

```
> x <- c(1, 3, 5, 2, 6)   # declaración del vector x
> x
[1] 1 3 5 2 6
> y <- 1:5                # declaración del vector y
> y
[1] 1 2 3 4 5
> y+2                     # suma escalar
[1] 3 4 5 6 7
> y*2                     # producto escalar
[1]  2  4  6  8 10
> y^2                     # potencia
[1]  1  4  9 16 25
> x+y                     # suma
[1]  2  5  8  6 11
> x*y                     # producto
[1]  1  6 15  8 30
> x/y                     # división
[1] 1.000000 1.500000 1.666667 0.500000 1.200000
> sum(x)                  # suma de de los elementos de x
[1] 17
> sum(x+y)                # suma de los elementos de x+y
[1] 32
```

Una expresión de R puede incluir paréntesis, llamadas a funciones, asignaciones a variables. La siguiente tabla detalla los operadores de R.

| Operador | Descripción |
|---|---|
| - | Resta, operador unario o binario |
| + | Suma, operador unario o binario |
| ! | Negación, operador unario |
| ~ | Tilde para fórmulas, operador unario o binario |
| ? | Ayuda |
| : | Secuencia, operador binario |

| Operador | Descripción |
|---|---|
| * | Producto, operador binario |
| / | División, operador binario |
| ^ | Potencia, operador binario |
| %x% | Operador binario especial, x puede remplazarse |
| %% | Módulo, operador binario |
| %/% | División entera, operador binario |
| %*% | Producto de matrices, operador binario |
| %o% | Producto exterior, operador binario |
| %x% | Producto Kronecker, operador binario |
| %in% | Comparación, operador binario |
| < | Menor que |
| > | Mayor que |
| == | Igual |
| >= | Mayor o igual que |
| <= | Menor o igual que |
| & | And |
| && | And, no vectorizado |
| \| | Or |
| \|\| | Or, no vectorizado |
| <- | Asignación al lado izquierdo |
| -> | Asignación al lado derecho |
| $ | Subconjunto de una lista, operador binario |

### 2.1.3. El espacio de trabajo y los objetos

R crea objetos para manipular datos. Los objetos pueden ser variables, arrays de números, cadenas de caracteres, funciones o estructuras más complejas construidas a partir de estos objetos. Durante una sesión de trabajo en R se crea un espacio de trabajo que almacena los objetos. El comando `objects()` muestra los objetos creados durante la sesión. Para eliminar un objeto del espacio de trabajo se utiliza el comando `rm(object)`. Si se desea eliminar todos los objetos, se debe ejecutar `rm(list=ls())`.

## 2.2. Tipos de datos

En R las variables no se declaran y su tipo de dato queda determinado por los valores que almacena. Por ejemplo, si se asigna una secuencia de números a la variable `x`, entonces se convierte en un vector de números. Dada una variable `x`, la función `class(x)` devuelve la clase a la que pertenece e indica las funciones que se pueden aplicar a la variable.

De forma general, a las variables se les denomina objetos. Los principales tipos de objetos de R son: vector, listas, `data.frame` y factor. Un vector es un conjunto de números, valores lógicos o caracteres; una lista es un conjunto de objetos; un factor es un conjunto clasificado en categorías y un `data.frame` es una tabla de datos.

El tipo de dato básico de R es un vector, un conjunto indexado de variables del mismo tipo. Un vector puede almacenar valores de tipo `integer`, `numeric`, `character`, `complex` y `logical`. Los valores almacenados en un vector se pueden etiquetar utilizando nombres. Por ejemplo,

```
> v <- c(n1=5, n2=3, n3=4, n4=6, n5=10)
> v
n1 n2 n3 n4 n5
 5  3  4  6 10
> sort(v)
n2 n3 n1 n4 n5
 3  4  5  6 10
> names(v)
[1] "n1" "n2" "n3" "n4" "n5"
```

### 2.2.1. Numeric

Un valor numérico en R puede ser de tipo `integer` o `double`. La función `format()` modifica el formato de un valor numérico.

```
> format(pi, scientific=TRUE)
[1] "3.141593e+00"
> format(c(1, 10, 100, 1000), trim=FALSE)
[1] "   1" "  10" " 100" "1000"
> format(c(1, 10, 100, 1000), trim=TRUE)
[1] "1"    "10"   "100"  "1000"
```

### 2.2.2. Logical

Los valores lógicos almacenan los valores falso y verdadero, representados por `TRUE` y `FALSE`, aunque también se puede utilizar `T` y `F`, respectivamente. El siguiente ejemplo muestra la evaluación de una expresión lógica para todos los elementos de un vector,

```
> x <- 1:10
> (x%%2==0)
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE

> any(x>=10)
[1] TRUE
> all(x>5)
[1] FALSE
> y <- c(2, 4, 6, 8, 10, 12, 14, 16, 18, 20)
> x %in% y
 [1] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
> x[x %in% y]
[1]  2  4  6  8 10
```

Los operadores lógicos de R ofrece se aplican a vectores y objetos de tipo lógico. Si los operandos son vectores, el resultado es un vector.

| Operador | Descripción |
|---|---|
| ¡x | Negación de x |
| x & y | x AND y, devuelve un vector |
| x && y | x AND y, devuelve un solo valor |
| x \| y | x OR y, devuelve un vector |
| x \|\| y | x OR y, devuelve un solo valor |
| xor(x, y) | OR exclusivo |
| x %in% y | x IN y |
| x < y | x menor que y |
| x > y | x mayor que y |
| x <= y | x menor o igual que y |
| x >= y | x mayor o igual que y |
| x == y | x igual que y |
| x ¡= y | x distinto de y |

R ofrece funciones específicas para vectores y objetos de tipo `logical`.

| Función | Descripción |
|---|---|
| isTRUE(x) | Devuelve TRUE si todos los valores de x son TRUE |
| all(...) | Devuelve TRUE si todos los argumentos son TRUE |
| any(...) | Devuelve TRUE si al menos un argumento es TRUE |
| identical(x, y) | Compara dos objetos y devuelve TRUE si son iguales |
| all.equal(x, y) | Comprueba si dos objetos son iguales |

### 2.2.3. Character

Los caracteres y las cadenas de caracteres se delimitan utilizando apóstrofes o comillas, es válido utilizar la expresión ‘a’ o “a” para asignar un valor de tipo `character`. R ofrece funciones de concatenación de cadenas de caracteres, extracción de cadenas y búsqueda de patrones dentro de cadenas.

```
> cat("Hola", "mundo", "\n")
Hola mundo
> cat("a","e","i","o","u", "\n")
a e i o u
> cat("a","e","i","o","u", sep=",", "\n")
a,e,i,o,u,
> paste("Hola","mundo", "\n")
[1] "Hola mundo \n"
> cat(paste("Hola","mundo", "\n"))
Hola mundo
> print(paste("Hola","mundo", "\n"))
[1] "Hola mundo \n"
> substr("Hola, mundo", 1, 4)
[1] "Hola"
> nchar(c("lunes", "martes", "miercoles", "jueves", "viernes"))
[1] 5 6 9 6 7
```

```
> tolower(LETTERS)
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
[20] "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"
> noquote(letters)
 [1] a b c d e f g h i j k l m n o p q r s t u v w x y z
> noquote(sub("a", "A", letters))
 [1] A b c d e f g h i j k l m n o p q r s t u v w x y z
```

Las funciones de búsqueda de patrones utilizan expresiones regulares y símbolos especiales como `^ $ . .{n}` o `[ch1-ch2]` para describir el patrón de búsqueda. El significado de cada símbolo se describe en la siguiente tabla.

| Expresión | Descripción |
|---|---|
| ^ | Inicio de cadena |
| $ | Fin de cadena |
| . | Cualquier carácter |
| .{n} | Cualquier cadena de caracteres de longitud n |
| [ch1-ch2] | Rango de caracteres desde ch1 hasta ch2 |
| [ch1,ch2,ch3] | Conjunto de caracteres ch1, ch2 y ch3 |

Por ejemplo, la función `colors()` devuelve los nombres de los 657 colores definidos en R. Para seleccionar los colores que contienen la cadena `"sky"` se utiliza la expresión `colors()[grep("sky", colors())]`.

```
 > colors()[grep("sky", colors())]
  [1] "deepskyblue"   "deepskyblue1"  "deepskyblue2"  "deepskyblue3"
  [5] "deepskyblue4"  "lightskyblue"  "lightskyblue1" "lightskyblue2"
  [9] "lightskyblue3" "lightskyblue4" "skyblue"       "skyblue1"
[13] "skyblue2"      "skyblue3"      "skyblue4"
```

Si se desea seleccionar los colores que comienzan por "`sky`" se utiliza la expresión `colors()[grep("^sky", colors())]`.

```
 > colors()[grep("^sky", colors())]
[1] "skyblue"  "skyblue1" "skyblue2" "skyblue3" "skyblue4"
```

Para seleccionar los colores que terminan con "`blue`" se utiliza la expresión `colors()[grep("blue$", colors())]`.

```
> colors()[grep("blue$", colors())]
 [1] "aliceblue"  "blue"           "cadetblue"      "cornflowerblue"
 [5] "darkblue"   "darkslateblue"  "deepskyblue"    "dodgerblue"
 [9] "lightblue"  "lightskyblue"   "lightslateblue" "lightsteelblue"
[13] "mediumblue" "mediumslateblue" "midnightblue"  "navyblue"
[17] "powderblue" "royalblue"      "skyblue"        "slateblue"
[21] "steelblue"
```

Si se desea seleccionar los colores que contienen "`red`" y a continuación hay un carácter se utiliza la expresión `colors()[grep("red.", colors())]`.

```
> colors()[grep("red.", colors())]
 [1] "indianred1"     "indianred2"     "indianred3"     "indianred4"
 [5] "orangered1"     "orangered2"     "orangered3"     "orangered4"
 [9] "palevioletred1" "palevioletred2" "palevioletred3" "palevioletred4"
[13] "red1"           "red2"           "red3"           "red4"
[17] "violetred1"     "violetred2"     "violetred3"     "violetred4"
```

Por último, para seleccionar los colores que comienzan por "r" o "t" se utiliza la expresión `colors()[grep("^[r,t]", colors())]`.

```
> colors()[grep("^[r,t]", colors())]
 [1] "red"        "red1"       "red2"       "red3"       "red4"
 [6] "rosybrown"  "rosybrown1" "rosybrown2" "rosybrown3" "rosybrown4"
[11] "royalblue"  "royalblue1" "royalblue2" "royalblue3" "royalblue4"
[16] "tan"        "tan1"       "tan2"       "tan3"       "tan4"
[21] "thistle"    "thistle1"   "thistle2"   "thistle3"   "thistle4"
[26] "tomato"     "tomato1"    "tomato2"    "tomato3"    "tomato4"
[31] "turquoise"  "turquoise1" "turquoise2" "turquoise3" "turquoise4"
```

La siguiente tabla describe las funciones de uso común para cadenas de caracteres.

| Función | Descripción |
|---|---|
| cat(...) | Concatena los objetos, separados por un espacio y los imprime por la consola |
| paste(...) | Concatena los objetos y devuelve una cadena de caracteres |
| print(x) | Imprime un objeto |
| substr() | Extrae una cadena de un vector de caracteres |
| strtrim() | Elimina los espacios de un vector de caracteres |
| strsplit() | Divide los objetos de un vector de caracteres utilizando un carácter como delimitador |
| grep() | Busca coincidencias con un patrón dentro de un vector de caracteres |
| grepl() | Busca coincidencias con un patrón dentro de un vector de caracteres y devuelve un vector lógico |
| agrep() | Similar a grep(), busca coincidencias aproximadas |
| gsub(p, r, v) | Reemplaza todas las ocurrencias del patrón p por r en un vector de caracteres |
| sub(p, r, v) | Reemplaza la primer ocurrencia del patrón p por r en un vector de caracteres |
| tolower(x) | Convierte x a minúsculas |
| toupper(x) | Convierte x a mayúsculas |
| noquote(x) | Imprime un vector de caracteres sin comillas |
| nchar(x) | Número de caracteres |
| letters | Vector de letras minúsculas |
| LETTERS | Vector de letras mayúsculas |

### 2.2.4. Factor

Un `factor` se utiliza para almacenar un conjunto finito de valores. El tipo `factor` es útil para almacenar información definida por un conjunto de valores. El sexo o el estado civil de una persona son ejemplos de uso de un `factor`, donde el sexo se define por el conjunto {'M', 'F'} y el estado civil por {'S', 'C', 'D', 'V'}. La función `factor()` codifica un vector como factor. El atributo `levels` muestra los valores únicos del conjunto.

```
> sexo <- c("M", "F")
> is.factor(sexo)
[1] FALSE
> sexo <- factor(sexo)
> sexo
[1] M F
Levels: F M
> estado <- factor(c("S", "C", "D", "V"))
> estado
[1] S C D V
Levels: C D S V
> nlevels(estado)
[1] 4
```

La siguiente tabla describe las funciones de uso común para objetos de tipo `factor`.

| Función | Descripción |
|---|---|
| levels(x) | Devuelve el conjunto de niveles de x |
| nlevels(x) | Devuelve el número de niveles de x |
| relevel(x, ref) | Reordena los niveles de x, empezando por ref |

### 2.2.5. Date

Los objetos `date` de R tienen el formato año-mes-día. La función `Sys.Date()` devuelve la fecha actual. La función `as.Date()` convierte una cadena de caracteres en un objeto de tipo `date` de acuerdo con el formato que se indique.

| Expresión | Descripción |
|---|---|
| %a | Nombre abreviado del día |
| %A | Nombre completo del día |
| %d | Día del mes |
| %b | Nombre abreviado del mes |
| %B | Nombre completo del mes |
| %m | Número correspondiente al mes |
| %y | Año de dos dígitos |
| %Y | Año de cuatro dígitos |

Para convertir una cadena de caracteres a un objeto de tipo `date` es necesario indicar el formato de la fecha.

```
> as.Date("01-01-2013", format="%d-%m-%Y")
[1] "2013-01-01"
> as.Date("01-ene-2013", format="%d-%b-%Y")
[1] "2013-01-01"
> as.Date("01ene2013", format="%d%b%Y")
[1] "2013-01-01"
> format.Date(as.Date("01-01-2013", format="%d-%m-%Y"), "%d-%m-%Y")
[1] "01-01-2013"
```

Para calcular automáticamente una secuencia de fechas se utiliza la función `seq.Date(from, to, by, length.out = NULL)`. Los argumentos `from` y `to` son de tipo `date`, indican la fecha de inicio y la fecha de fin, respectivamente. El argumento `by` define el intervalo entre cada fecha y toma los valores `"day"`, `"week"`, `"month"` o `"year"`. También se puede utilizar el plural de estas etiquetas y añadir un número para expresar una cantidad, por ejemplo `"7 days"`, o `"4 weeks"`. El argumento `length.out` es un valor numérico que indica la longitud de la secuencia.

```
> ene <- seq.Date(as.Date("2013/01/01"), as.Date("2013/01/31"),
+ by="day")
> ene
 [1] "2013-01-01" "2013-01-02" "2013-01-03" "2013-01-04" "2013-01-05"
 [6] "2013-01-06" "2013-01-07" "2013-01-08" "2013-01-09" "2013-01-10"
[11] "2013-01-11" "2013-01-12" "2013-01-13" "2013-01-14" "2013-01-15"
[16] "2013-01-16" "2013-01-17" "2013-01-18" "2013-01-19" "2013-01-20"
[21] "2013-01-21" "2013-01-22" "2013-01-23" "2013-01-24" "2013-01-25"
[26] "2013-01-26" "2013-01-27" "2013-01-28" "2013-01-29" "2013-01-30"
[31] "2013-01-31"
> feb <- seq.Date(as.Date("2013/02/01"), by="weeks", length.out=4)

> feb
[1] "2013-02-01" "2013-02-08" "2013-02-15" "2013-02-22"
> año <- seq.Date(as.Date("2013/01/01"), by="months", length.out=12)
> año
 [1] "2013-01-01" "2013-02-01" "2013-03-01" "2013-04-01" "2013-05-01"
 [6] "2013-06-01" "2013-07-01" "2013-08-01" "2013-09-01" "2013-10-01"
[11] "2013-11-01" "2013-12-01"
```

La siguiente tabla describe las funciones de uso común para objetos de tipo `date`.

| Función | Descripción |
|---|---|
| Sys.Date() | Devuelve la fecha actual |
| as.Date() | Convierte una cadena de caracteres en date |
| format.Date | Modifica el formato de la fecha |
| seq.Date() | Calcula una secuencia de fechas |
| cut.Date() | Divide fechas en intervalos |
| julian | Calcula los días transcurridos desde una fecha |

### 2.2.6. NA, NaN, NULL

R utiliza el valor `NA`[6] para indicar cuando no dispone de un valor para un objeto. Si el tipo de dato del objeto es numérico y no se dispone de un valor, entonces R utiliza `NaN`[7].

El objeto `NULL` indica la ausencia de un objeto. No debe confundirse con el valor de un vector o una lista de cero elementos. El objeto `NULL` no tiene tipo. Solo existe una instancia de este objeto en R y a ella se refieren todos los objetos que no están inicializados. Cuando se realiza una operación aritmética que produce un valor no numérico se representa como `NaN`. Esto se puede dar al dividir entre cero o realizar operaciones con el valor infinito.

```
> 0/0
[1] NaN
> Inf / Inf
[1] NaN
```

Para comprobar si un objeto almacena valores `NA`, `NaN` o `NULL`, se utilizan las funciones `is.na(x)`, `is.nan(x)` e `is.null(x)`, respectivamente.

```
> x <- c(1, 2, 8, 0, NA)
> is.na(x)
[1] FALSE FALSE FALSE FALSE  TRUE
> y <- x/0
> y
[1] Inf Inf Inf NaN  NA
> is.nan(y)
[1] FALSE FALSE FALSE  TRUE FALSE
> is.na(y)
[1] FALSE FALSE FALSE  TRUE  TRUE
```

### 2.2.7. Conversión de tipos de datos

Todos los objetos de R tienen un tipo de dato. Para conocer el tipo de cualquier objeto se utiliza la función `typeof(x)`.

```
> x <- c(1, 2, 8, 0, NA)
> typeof(x)
[1] "double"
```

R ofrece funciones para comprobar cada tipo primitivo del lenguaje y para convertir un objeto a un tipo de dato determinado.

---

[6] NA (Not Available)
[7] NAN (Not a Number) representa un valor que no se puede definir, como la división de un número entre cero.

| Tipo de dato | Comprobación | Conversión |
|---|---|---|
| array | is.array() | as.array() |
| character | is.character() | as.character() |
| data.frame | is.data.fame() | as.data.fame() |
| factor | is.factor() | as.factor() |
| list | is.list() | as.list() |
| logical | is.logical() | as.logical() |
| matrix | is.matrix() | as.matrix() |
| numeric | is.numeric() | as.numeric() |
| vector | is.vector() | as.vector() |

### 2.2.8. La función typeof()

La función `typeof(object)` de R devuelve el tipo de un objeto de R. El tipo de objeto determina la estructura de los datos y los elementos que la componen. Los objetos de R representan, en realidad, punteros a estructuras de datos desarrolladas en el código C subyacente a R. La siguiente tabla muestra los valores de retorno de uso común de la función `typeof(object)`.

| Valor de typeof | Descripción |
|---|---|
| "NULL" | NULL |
| "symbol" | El nombre de una variable |
| "pairlist" | Objeto de uso interno denominado 'pairlist' |
| "closure" | Función |
| "environment" | Entorno |
| "promise" | Objeto que implementa una "evaluación perezosa" |
| "language" | Constructor del lenguaje |
| "special" | Función interna que no evalúa sus argumentos |
| "builtin" | Función interna que evalúa sus argumentos |
| "char" | Objeto interno de tipo cadena de caracteres |
| "logical" | Vector de valores lógicos |
| "integer" | Vector de números enteros |
| "double" | Vector de números reales |
| "complex" | Vector de números complejos |
| "character" | Vector de caracteres |
| "..." | Argumento de longitud variable |
| "any" | Tipo especial que almacena todos los tipos |
| "expression" | Objeto de tipo expresión |
| "list" | Lista |
| "bytecode" | Bytecode de uso interno |
| "externalprt | Objeto de tipo puntero externo |
| "weakref" | Objeto de referencia débil |

| Valor de typeof | Descripción |
|---|---|
| "raw" | Vector de bytes |
| "S4" | Objeto S4 que no es un objeto simple |

## 2.3. Estructuras de datos

En todo lenguaje de programación, las variables representan el medio para acceder a los datos almacenados en la memoria. R ofrece un conjunto de estructuras de datos a las que se refiere de forma genérica como objetos o variables. Las estructuras de datos disponibles en R incluyen vectores, arrays, matrices, listas, expresiones y funciones. Estas estructuras pueden almacenar valores lógicos, cadenas de caracteres, números enteros, números reales y números complejos, representados por los tipos de datos de R: `logical`, `character`, `integer`, `double` y `complex`.

Durante el proceso de cálculo, los objetos de R se convierten automáticamente a otros tipos de datos para asegurar que los datos con compatibles entre sí y poder garantizar que los cálculos se realizan correctamente.

### 2.3.1. Vector

Un vector es un conjunto ordenado de objetos del mismo tipo. La función `c()`es muy útil para inicializar vectores. Esta función concatena la lista de argumentos que recibe y los transforma en un vector.

```
> x <- c(9.0, 5.5, 4.5, 6.0, 7.5)
```

El operador de asignación almacena en la variable `x` el resultado de ejecutar la función `c(9.0, 5.5, 4.5, 6.0, 7.5).` El valor de retorno es el vector que se obtiene de la concatenación de los argumentos de la función. Para mostrar el contenido del vector `x` en la consola basta con introducir el identificador del vector para que R muestre los valores almacenados. La función `assign()` es equivalente al operador de asignación y permite asignar un valor a un objeto. Por ejemplo, la inicialización del vector `x` también se puede realizar de la siguiente forma:

```
> assign("x", c(9.0, 5.5, 4.5, 6.0, 7.5))
> x
[1] 9.0 5.5 4.5 6.0 7.5
```

La función `c()` admite argumentos de cualquier tipo. Si en vez de un conjunto de números se utiliza un conjunto de caracteres, entonces la función devuelve un vector de caracteres.

```
> ch <- c("a", "e", "i", "o", "u")
```

Si x1 se inicializa con los valores {16, 8, 4, 2, 1} y x2 con {2, 4, 8, 16}, el vector x3 se puede definir a partir de los vectores x1 y x2 para representar potencias de 2.

```
> assign("x1", c(16, 8, 4, 2, 1))
> assign("x2", c(2, 4, 8, 16))
> assign("x3", c(x1, 0, 1/x2))
> x1
[1] 16  8  4  2  1
> x2
[1]  2  4  8 16
> x3
[1] 16.0000  8.0000  4.0000  2.0000  1.0000  0.0000  0.5000  0.2500  0.1250
[10]  0.0625
```

La función seq() facilita la definición de secuencias de números. Esta función tiene cinco argumentos: from, to, by, length y along. Los argumentos from y to especifican el inicio y el fin de la secuencia. Estos valores se pueden especificar por su nombre o indicarse de forma implícita. Por ejemplo, seq(from=1, to=10) es equivalente a seq(1, 10). Ambas secuencias son equivalentes al vector 1:10. Los argumentos by y length definen el incremento y la longitud de la secuencia, respectivamente. El valor por defecto del incremento es 1. Por ejemplo, la expresión seq(-2, 2, by=0.5) define el vector {-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5, 2}. Este vector también se puede definir como seq(length=9, from=-2, by=0.5).

```
> x <- seq(-2, 2, by=0.5)
> x
[1] -2.0 -1.5 -1.0 -0.5 0.0 0.5 1.0 1.5 2.0
> y <- seq(length=9, from=-2, by=0.5)
> y
[1] -2.0 -1.5 -1.0 -0.5 0.0 0.5 1.0 1.5 2.0
```

El argumento along se utiliza para definir un vector que comienza en 1 y tiene tantos elementos como la longitud del vector indicado. La expresión seq(along=x) define el vector {1, 2, 3, 4, 5, 6, 7, 8, 9}. Si el total de elementos del vector que se utiliza como argumento está vacío, el resultado es una secuencia vacía.

```
> z <- seq(along=x)
> z
[1] 1 2 3 4 5 6 7 8 9
```

La función rep() se utiliza para replicar un objeto un número determinado de veces. Por ejemplo, la expresión rep(x, times=2) define un vector que almacena dos vectores x. Si se utiliza el argumento each, entonces los elementos del vector original se repiten el total de veces indicadas antes de pasar al siguiente elemento.

```
> x2 <- rep(x, times=2)
> x2
[1] -2.0 -1.5 -1.0 -0.5 0.0 0.5 1.0 1.5 2.0 -2.0 -1.5   -1.0 -0.5 0.0 0.5
1.0 1.5 2.0
> x3 <- rep(x, each=2)
```

```
> x3
[1] -2.0 -2.0 -1.5 -1.5 -1.0 -1.0 -0.5 -0.5 0.0 0.0 0.5  0.5 1.0 1.0 1.5 1.5
2.0 2.0
```

R facilita la manipulación de vectores que almacenan valores numéricos y lógicos. Los valores que almacena un vector lógico son TRUE, para verdadero, FALSE para falso y NA para un valor que no se ha inicializado. Un vector lógico se inicializa cuando al aplicar una condición que da como resultado un valor falso o verdadero. Por ejemplo, la expresión `x >= 0` define un vector donde sus elementos almacenan TRUE si el valor del elemento de x es mayor o igual a cero y FALSE en cualquier otro caso.

```
> x
[1] -2.0 -1.5 -1.0 -0.5 0.0 0.5 1.0 1.5 2.0
> x1 <- x >=0
> x1
[1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE
```

En una condición lógica se pueden utilizar operadores de comparación, de igualdad y el operador distinto de. Los operadores de comparación se definen por los lexemas "<", "<=", ">", ">=" para las condiciones menor que, menor o igual, mayor y mayor o igual, respectivamente. El operador de igualdad se define por el lexema "==" y el operador distinto de por "`!=`". El operador lógico "and" de una condición se representa con el símbolo "&" y el operador "or" con el símbolo "`|`".

La función `ifelse()` evalúa una condición para cada uno de los elementos de un vector. Por ejemplo, para saber si el valor numérico es mayor o igual a cero, se puede comprobar de la siguiente forma:

```
> x
[1] -2.0 -1.5 -1.0 -0.5 0.0 0.5 1.0 1.5 2.0
x1 <- ifelse(x>=0, TRUE, FALSE)
> x1
[1] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE
```

A un vector de tipo lógico se le puede aplicar un operador aritmético. En ese caso, el valor FALSE se considera 0 y el valor TRUE 1. Si el valor es NA, entonces no se puede realizar la operación aritmética porque se trata de un valor no conocido. En este caso, la especificación de la operación aritmética está incompleta. La función `is.na(x)` devuelve un vector lógico del mismo tamaño que el vector x indicando los valores de x que son NA. Por ejemplo, si se define un vector b con la expresión `c(1>2, 2>=1, NA)`, entonces la función `is.na(b)` devuelve FALSE, FALSE, TRUE.

```
> b <- c(1>2, 2>=1, NA)
> b
[1] FALSE TRUE NA
> is.na(b)
[1] FALSE FALSE TRUE
```

Si se ejecuta la expresión `b == NA`, esta condición devuelve un vector de tamaño 3 con todos los valores `NA` ya que la condición lógica está incompleta y no es decidible. Los valores del vector `b` permanecen sin cambio.

```
> b
[1] FALSE TRUE NA
> b == NA
[1] NA NA NA
```

Cuando se realiza una operación aritmética que produce un valor no numérico se representa como `NaN`. Esto se puede dar al dividir entre cero o realizar operaciones con el valor infinito. La función `is.na(x)` devuelve `TRUE` si los elementos del vector `x` son `NA` o `NaN`.

```
> b <- c(1>2, 0/0, NA)
> b
[1] 0 NaN NA
> is.na(b)
[1] FALSE TRUE TRUE
```

Los vectores de caracteres son útiles para almacenar etiquetas asociadas a los datos y otros valores de tipo alfanumérico. La función `c()` permite definir un vector de caracteres y la función `paste()` facilita la concatenación de sus argumentos para inicializar un vector. En el siguiente ejemplo, la función `paste()` concatena las letras de las vocales con su número ordinal aplicando expresamente el separador vacío, ya que el valor por defecto del separador es un espacio en blanco.

```
> ch <- c("a", "e", "i", "o", "u")
> ch
[1] "a" "e" "i" "o" "u"
> ch1 <- paste(c("a", "e", "i", "o", "u"), 1:5, sep="")
> ch1
[1] "a1" "e2" "i3" "o4" "u5"
```

El tamaño de los vectores que se concatenan puede no coincidir. En ese caso, la función `paste()` repite la secuencia de elementos del vector de menor tamaño.

```
> ch2 <- paste(c("a", "e", "i", "o", "u"), 1:10, sep="")
> ch2
[1] "a1" "e2" "i3" "o4" "u5" "a6" "e7" "i8" "o9" "u10"
```

La función `which()` localiza los valores verdaderos de un vector de valores lógicos, `which.min()` localizar el valor mínimo y `which.max()` el máximo de un vector de valores numéricos.

```
> w <- c(4, 2, 7, 10, 1, 0)
> w >= 4
[1]  TRUE FALSE  TRUE  TRUE FALSE FALSE
> which(w >= 4)
[1] 1 3 4
> which.max(w)
[1] 4
```

```
> w[which.max(w)]
[1] 10
> which.min(w)
[1] 6
```

La función `match()` encuentra la primera posición de un elemento en un vector.

```
> w1 <- c(4, 2, 7, 2, 4, 10, 1, 0)
> w2 <- c(4, 2, 7, 10, 1, 0)
> match(w1, 2)
[1] NA  1 NA  1 NA NA NA NA
> match(w1, w2)
[1] 1 2 3 2 1 4 5 6
> match(w2, w1)
[1] 1 2 3 6 7 8
```

#### 2.3.1.1. Acceso a los elementos de un vector

Para referenciar los elementos de un vector se utilizan los corchetes `[]`. La posición de los elementos del vector se indica utilizando uno o más índices.

```
> x <- c(4, 2, 7, 10, 1, 0)
> x[3]
[1] 7
> x[1:4]
[1] 4 2 7 10
> x[c(1, 3, 5)]
[1] 4 7 1
> x[x>2]
[1] 4 7 10
> x[5] <- 9
> x
[1] 4 2 7 10 9 0
```

Para descartar elementos de un vector, se indican las posiciones y se utiliza un signo negativo. El resultado de la operación no se realiza sobre el vector, a menos que se realice una asignación.

```
> x
[1] 4 2 7 10 9 0
> x[-3]
[1] 4 2 10 9 0
> x
[1] 4 2 7 10 9 0
> x <- x[-c(1, 3)]
> x
[1] 2 10 9 0
```

#### 2.3.1.2. Operaciones con vectores

Las operaciones matemáticas entre vectores se calculan elemento a elemento.

```
> x
[1] 2 10 9 0
> x2 <- 2*x^2
> x2
[1] 8 200 162 0
> x + x2
[1] 10 210 171 0
```

#### 2.3.1.3. Funciones de uso común

A continuación se describen las funciones de uso común con vectores.

| Función | Descripción |
|---|---|
| sum(x) | Suma los elementos de x |
| prod(x) | Producto de los elementos de x |
| cumsum(x) | Suma acumulativa de los elementos de x |
| cumprod(x) | Producto acumulativo de los elementos de x |
| min(x) | Valor mínimo de x |
| max(x) | Valor máximo de x |
| mean(x) | Media de x |
| median(x) | Mediana de x |
| var(x) | Varianza de x |
| sd(x) | Desviación estándar de x |
| cov(x, y) | Covarianza de x, y |
| cor(x, y) | Correlación de x, y |
| range(x) | Rango de x |
| quantile(x) | Cuantiles de x |
| diff(x) | Diferencia entre los elementos i+1 e i del vector |
| diff(x, n) | Diferencia entre los elementos i+n e i del vector |
| fivenum(x) | Resumen de cinco números de x |
| length(x) | Número de elementos de x |
| unique(x) | Elementos únicos de x |
| rev(x) | Orden inverso de los elementos de x |
| sort(x) | Ordena los elementos de x |
| which(x) | Indices TRUE de un vector de valores lógicos |
| which.max(x) | Indice del valor Max de un vector |
| which.min(x) | Indice del valor Min de un vector |
| match(x, v) | Primera posición de un elemento de x con valor v |
| union(x, y) | Unión de x, y |
| intersect(x, y) | Intersección de x, y |
| setdiff(x, y) | Elementos de x que no están en y |
| setequal(x, y) | Indica si ambos vectores tienen los mismos elementos |

### 2.3.2. Matriz

Una matriz es un vector de dos dimensiones. Para crear una matriz se utiliza la función `matrix(data=NA, nrow=1, ncol=1, byrow=FALSE, dimnames=NULL)`. El argumento `data` es un vector que almacena los datos con los que se va a inicializar la matriz, `nrow` indica el número de filas, `ncol` indica el número de columnas, `byrow` determina si la matriz se debe rellenar por columnas o por, `dimnames` es una lista que contiene las etiquetas de las filas y las columnas. El valor por defecto de `byrow` es

FALSE, de manera que la matriz se rellena por columnas a menos que se indique lo contrario. La lista de las etiquetas de las filas y las columnas es un argumento opcional.

El siguiente ejemplo define las matrices `m1` y `m2`, dos cuadrados mágicos que suman 15 en cada fila y columna.

```
> m1 <- matrix(c(2, 7, 6, 9, 5, 1, 4, 3, 8), nrow=3, ncol=3,
+ byrow=TRUE,
+ dimnames=list(rows=c("1", "2", "3"), cols=c("1", "2", "3")))
> m2 <- matrix(c(2, 7, 6, 9, 5, 1, 4, 3, 8), nrow=3, ncol=3,
+ byrow=FALSE,
+ dimnames=list(rows=c("1", "2", "3"), cols=c("1", "2", "3")))
> m1
    cols
rows 1 2 3
   1 2 7 6
   2 9 5 1
   3 4 3 8
> m2
    cols
rows 1 2 3
   1 2 9 4
   2 7 5 3
   3 6 1 8
```

#### 2.3.2.1. Acceso a los elementos de una matriz

De forma similar a los vectores, para referenciar los elementos de una matriz se utilizan los corchetes `[]`. La posición de los elementos de la matriz se indica utilizando uno o más índices.

```
> m3 <- matrix(c(2, 7, 6, 9, 5, 1, 4, 3, 8), nrow=3, ncol=3)
> m3
     [,1] [,2] [,3]
[1,]    2    9    4
[2,]    7    5    3
[3,]    6    1    8
> m3[3,3]
[1] 8
> m3[1, ]
[1] 2 9 4
> m3[, 1]
[1] 2 7 6
> m3[c(1,3), ]
     [,1] [,2] [,3]
[1,]    2    9    4
[2,]    6    1    8
```

Además de referenciar a los elementos de una matriz por la posición de sus elementos, también se pueden utilizar los nombres de las filas y las columnas.

```
> m4 <- matrix(c(2, 7, 6, 9, 5, 1, 4, 3, 8), nrow=3, ncol=3,
+ dimnames=list(rows=c("f1", "f2", "f3"), cols=c("c1", "c2", "c3")))
> m4
    cols
rows c1 c2 c3
  f1  2  9  4
  f2  7  5  3
  f3  6  1  8
```

```
> m4["f1", ]
c1 c2 c3
 2  9  4
> m4[, "c2"]
f1 f2 f3
 9  5  1
> m4["f3", "c3"]
[1] 8
```

#### 2.3.2.2. Operaciones con matrices

La función `apply(X, MARGIN, FUN)` permite ejecutar funciones sobre elementos de una matriz. El argumento `X` representa la matriz, `MARGIN` indica si la función se aplica a las filas `(MARGIN=1)`, a las columnas `(MARGIN=2)` o a una matriz. `FUN` es el nombre de la función. Por ejemplo, se puede aplicar la función `sum` para calcular la suma para cada fila o columna.

```
> m5 <- matrix(1:9, nrow=3, ncol=3,
+ dimnames=list(rows=c("f1", "f2", "f3"), cols=c("c1", "c2", "c3")))
> m5
    cols
rows c1 c2 c3
  f1  1  4  7
  f2  2  5  8
  f3  3  6  9
> apply(m5, 1, sum)
f1 f2 f3
12 15 18
> apply(m5, 1, mean)
f1 f2 f3
 4  5  6
> apply(m5, 2, sum)
c1 c2 c3
 6 15 24
```

La función `apply(X, MARGIN, FUN)` admite como argumento una función personalizada. Por ejemplo, si se desea calcular la suma para cada fila de la matriz, se puede utilizar la expresión `apply(m5, 1, function(x) {sum(x)})`, esto sería equivalente a `apply(m5, 1, sum)`. El uso de funciones personalizadas aporta potencia de cálculo. Por ejemplo, si se desea calcular la media de los vectores de la matriz, se puede utilizar la expresión `apply(m5, 1, function(x) {sum(x)/length(x)})`, equivalente a `apply(m5, 1, mean)`.

```
> apply(m5, 1, function(x) { sum(x) })
f1 f2 f3
12 15 18

> apply(m5, 1, function(x) { sum(x)/2 })
 f1  f2  f3
6.0 7.5 9.0

> apply(m5, 1, mean)
f1 f2 f3
 4  5  6
> apply(m5, 1, function(x) { sum(x)/length(x) })
f1 f2 f3
 4  5  6
```

Las operaciones matemáticas entre matrices se calculan elemento a elemento. Para realizar un producto entre dos matrices se utiliza el operador %*%.

```
> A <- matrix(c(1, 2, 3, 4), nrow=2, ncol=2)
> A
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> B <- matrix(2:5, nrow=2, ncol=2)
> B
     [,1] [,2]
[1,]    2    4
[2,]    3    5
> A*B
     [,1] [,2]
[1,]    2   12
[2,]    6   20


> A%*%B
     [,1] [,2]
[1,]   11   19
[2,]   16   28
```

### 2.3.2.3. Funciones de uso común

A continuación se describen las funciones de uso común con matrices.

| Función | Descripción |
|---|---|
| t(A) | Transpuesta de A |
| det(A) | Determinante de A |
| solve(A, b) | Resuelve la ecuación Ax=b para x |
| solve(A) | Matriz inversa de A |
| eigen(A) | Eigenvalores y eigenvectores de A |
| chol(A) | Factorización Choleski de A |
| diag(n) | Crea una matriz identidad de nxn |
| diag(A) | Elementos de la diagonal de A |
| diag(x) | Crea una matriz diagonal a partir del vector x |
| lower.tri(A) | Matriz triangular inferior |
| upper.tri(A) | Matriz triangular superior |
| apply() | Aplica una función a la matriz |
| rbind(...) | Combina los argumentos por filas |
| cbind(...) | Combina los agumentos por columnas |
| dim(A) | Dimensión de A |
| nrow(A) | Número de filas |
| ncol(A) | Número de columnas |
| colnames(A) | Nombres de las columnas de A |
| rownames(A) | Nombres de las filas de A |
| dimnames(A) | Dimensión de los nombres de A |

### 2.3.3. Array

Un array es un vector multidimensional. Para crear un array se utiliza la función `array(data=NA, dim=length(data), dimnames=NULL)`. El argumento `data` es un vector que almacena los datos con los que se va a inicializar el array, `dim` es un vector de uno o más elementos que indica las dimensiones del array, `dimnames` es una lista que contiene las etiquetas de cada una de las dimensiones del array.

Los elementos de un array deben ser objetos del mismo tipo.

```
> w <- array(1:12, dim=c(2,3,2), dimnames=list(c("f1","f2"),
+ c("c1","c2","c3"), c("a","b")))
> w
, , a

   c1 c2 c3
f1  1  3  5
f2  2  4  6

, , b

   c1 c2 c3
f1  7  9 11
f2  8 10 12
```

#### 2.3.3.1. Acceso a los elementos de un array

De forma similar a los vectores y a las matrices, para referenciar los elementos de un array se utilizan los corchetes `[]`. La posición de los elementos del array se indica utilizando uno o más índices.

```
> w[2, 3, 1]
[1] 6
> w[, "c2", ]
   a  b
f1 3  9
f2 4 10
> w[1,,]
   a  b
c1 1  7
c2 3  9
c3 5 11
> w[1:2,,"b"]
   c1 c2 c3
f1  7  9 11
f2  8 10 12
```

#### 2.3.3.2. Operaciones con arrays

La función `apply(X, MARGIN, FUN)` permite ejecutar funciones sobre elementos de un array. El argumento `X` representa el array, `MARGIN` indica si la función se aplica a las filas (`MARGIN=1`), a las columnas (`MARGIN=2`) o a una matriz. `FUN` es el nombre de la función. Por ejemplo, se puede aplicar la función `sum` para calcular la suma para cada fila o columna.

```
> apply(w, 1, sum)        # suma por filas
f1 f2
36 42
> apply(w, 2, sum)        # suma por columnas
c1 c2 c3
18 26 34
> apply(w, c(1,2), sum)   # suma por la dimensión 3: a, b
   c1 c2 c3
f1  8 12 16
f2 10 14 18

> apply(w, c(1,3), sum)   # suma por la dimensión 2: c1, c2, c3
    a  b
f1  9 27
f2 12 30
> apply(w, c(2,3), sum)   # suma por la dimensión 1: f1, f2
    a  b
c1  3 15
c2  7 19
c3 11 23
```

#### 2.3.3.3. Funciones de uso común

A continuación se describen las funciones de uso común con arrays.

| Función | Descripción |
|---|---|
| apply() | Aplica una función al array |
| aperm() | Transpone el array |
| dim(x) | Dimensión del array |
| dimnames(x) | Dimensión de los nombres del array |

### 2.3.4. Lista

Una lista es una colección de elementos. Este tipo de estructura de datos se puede considerar como el caso general de un vector, ya que los elementos de una lista no necesariamente almacenan el mismo tipo de dato. Cada elemento de una lista puede almacenar cualquier tipo de objeto de R. Para crear una lista se utiliza la función `list()`. La lista de argumentos que recibe tienen la forma `name=value`, aunque también se pueden especificar sin nombre.

```
> lista <- list(nombre="Carlos", notas=c(5,7,9,5,8), curso=2012)
> lista
$nombre
[1] "Carlos"

$notas
[1] 5 7 9 5 8

$curso
[1] 2012
```

La función `length()` devuelve el número total de elementos almacenados en una lista.

```
> vocales <- list("a", "e", "i", "o", "u")
```

```
> length(vocales)
[1] 5
```

#### 2.3.4.1. Acceso a los elementos de una lista

Para referenciar los elementos de una lista se utilizan los corchetes `[]`, los corchetes dobles `[[]]` o el símbolo `$`. Los elementos de la lista se pueden referenciar por su posición o por su nombre.

```
> lista[1]
$nombre
[1] "Carlos"
> lista[[1]]
[1] "Carlos"
> lista["nombre"]
$nombre
[1] "Carlos"
> lista[["nombre"]]
[1] "Carlos"
> lista$nombre
[1] "Carlos"
```

En una lista que almacena elementos de distinto tipo, es posible seleccionar un subconjunto de ellos.

```
> lista[1:2]
$nombre
[1] "Carlos"

$notas
[1] 5 7 9 5 8

> lista[c(1,2)]
$nombre
[1] "Carlos"

$notas
[1] 5 7 9 5 8

> lista[c(1,3)]
$nombre
[1] "Carlos"

$curso
[1] 2012
```

Si la lista se ha creado sin utilizar nombres, entonces solo es posible referenciar sus elementos por su posición. Por ejemplo, la expresión `lista[[1]]` accede al primer elemento de la lista.

```
> vocales <- list("a", "e", "i", "o", "u")
> vocales
[[1]]
[1] "a"

[[2]]
[1] "e"
```

```
[[3]]
[1] "i"

[[4]]
[1] "o"

[[5]]
[1] "u"
```

En este ejemplo, si se utiliza la expresión `vocales[1]`, se obtiene una lista, a diferencia de la expresión `vocales[[1]]`, que devuelve un carácter.

```
> vocales[1]
[[1]]
[1] "a"
> vocales[[1]]
[1] "a"
```

#### 2.3.4.2. Operaciones con listas

La función `lapply()` permite aplicar una función a cada elemento de una lista. Por ejemplo, se puede aplicar la función `seq` para definir cinco vectores distintos como elementos de la lista, para calcular la suma de los vectores o su media.

```
> lista <- lapply(1:5, seq)

> lista
[[1]]
[1] 1

[[2]]
[1] 1 2

[[3]]
[1] 1 2 3

[[4]]
[1] 1 2 3 4

[[5]]
[1] 1 2 3 4 5

> lapply(lista, sum)
[[1]]
[1] 1

[[2]]
[1] 3

[[3]]
[1] 6

[[4]]
[1] 10

[[5]]
[1] 15
```

```
> lapply(lista, mean)
[[1]]
[1] 1

[[2]]
[1] 1.5
[[3]]
[1] 2

[[4]]
[1] 2.5

[[5]]
[1] 3
```

#### 2.3.4.3. Funciones de uso común

A continuación se describen las funciones de uso común con listas.

| Función | Descripción |
|---|---|
| lapply() | Aplica una función a cada elemento de una lista y devuelve una lista |
| sapply() | Similar a lapply(), pero devuelve un vector o una matriz |
| vapply() | Similar a sApply(), pero incluye un tipo de dato de retorno predefinido |
| replicate() | Replica una lista |
| unlist(x) | Devuelve un vector con todos los elementos de x |
| length(x) | Número de elementos de x |
| names(x) | Nombres de los objetos de x |

### 2.3.5. Dataframe

Un `data.frame` es la estructura que se utiliza para almacenar matrices de datos en R. Un objeto `data.frame` es una lista de vectores, factores y otras matrices, en el que todos sus elementos deben tener el mismo número de filas. Un `data.frame` también se puede ver como una matriz en la que sus columnas pueden almacenar distintos tipos de objetos. El atributo `names` del `data.frame` permite etiquetar los datos almacenados. Los componentes de un `data.frame` deben ser de tipo vector, matriz de números, lista, `factor` o `data.frame`.

#### 2.3.5.1. Acceso a los elementos de un data.frame

Para referenciar a los elementos de un `data.frame` se pueden utilizar los corchetes `[]` o el símbolo `$`. Los corchetes se utilizan para referenciar filas y columnas y el `$` para una columna entera.

#### 2.3.5.2. Operaciones con objetos data.frame

La forma más simple de crear un `data.frame` es leer un fichero de datos con encabezados utilizando la función. `read.table(file, header=FALSE, sep="", skip, as.is, stringAsFactors=TRUE)`. El argumento `file` representa el nombre del fichero, `header` indica si la primera fila contiene los nombres de las columnas, `sep` es el carácter separados de los datos, `skip` contiene el número de filas que se deben omitir antes de leer los datos, `as.is` es un vector numérico que especifica las columnas que no se deben convertir en factores, `stringAsFactors` indica si los vectores de caracteres se deben convertir en factores.

En el siguiente ejemplo, el objeto `personas` es un `data frame` que almacena cinco columnas de datos distintas con valores de tipo cadena de caracteres y números.

```
> setwd("c:/Mis documentos de trabajo/R Data")
> personas <- read.table("personas.txt")
> personas
   Apellido1 Apellido2 Nombre Nacimiento Sueldo
01     López  González   Juan       1990   1200
02     Plata    Suárez   Luis       1991   1300
03   Sánchez Fernández  María       1986   1200
04    Torres     Ramos  Marta       1992   1000
05      Vega      Ríos  Sofía       1985   1400
> str(personas)
'data.frame':   5 obs. of  5 variables:
 $ Apellido1 : Factor w/ 5 levels "López","Plata",..: 1 2 3 4 5
 $ Apellido2 : Factor w/ 5 levels "del Río","Fernández",..: 3 5 2 4 1
 $ Nombre    : Factor w/ 5 levels "Juan","Luis",..: 1 2 3 4 5
 $ Nacimiento: int  1990 1991 1986 1992 1985
 $ Sueldo    : int  1200 1300 1200 1000 1400
```

Por defecto. R convierte las variables de tipo `String` en un `factor`. Para evitar esto, es necesario definir el argumento `stringsAsFactors=FALSE`.

```
> personas <- read.table("personas.txt", stringsAsFactors=FALSE)
> str(personas)
‘data.frame’:   5 obs. Of  5 variables:
 $ Apellido1 : chr  "López" "Plata" "Sánchez" "Torres" …
 $ Apellido2 : chr  "González" "Suárez" "Fernández" "Ramos" …
 $ Nombre    : chr  "Juan" "Luis" "María" "Marta" …
 $ Nacimiento: int  1990 1991 1986 1992 1985
 $ Sueldo    : int  1200 1300 1200 1000 1400
```

La función `read.csv(file, header=FALSE, sep=",", skip, as.is, stringAsFactors=TRUE)` es similar a `read.table()` y permite leer un fichero en

formato CSV[8] donde las columnas están separadas por una coma o por algún otro carácter.

```
> personas <- read.csv("personas-csv.txt", sep=";")
> personas
   Apellido1 Apellido2 Nombre Nacimiento Sueldo
01     López  González   Juan       1990   1200
02     Plata    Suárez   Luis       1991   1300
03   Sánchez Fernández  María       1986   1200
04    Torres     Ramos  Marta       1992   1000
05      Vega      Ríos  Sofía       1985   1400
```

Una vez que se ha creado el `data.frame` se pueden ejecutar funciones para analizar los datos.

```
> mean(personas$Sueldo)
[1] 1220
> max(personas$Nacimiento)
[1] 1992
```

Si se trabaja con un solo `data.frame`, se puede ejecutar la función `attach()` para hacer referencia a los nombres de las columnas del `data.frame` sin necesidad de indicar el nombre del `data.frame`. Al finalizar el análisis de los datos se debe hacer `detach()`.

```
> attach(personas)
> mean(Sueldo)
[1] 1220
> max(Sueldo)
[1] 1400
> min(Nacimiento)
[1] 1985
> detach(personas)
```

Una alternativa a las funciones `attach()` y `detach()` es el uso de `with()`. Esta función facilita el tratamiento de los datos.

```
> with(personas, mean(Sueldo))
[1] 1220
> with(personas, table(Nacimiento, Sueldo))
          Sueldo
Nacimiento 1000 1200 1300 1400
      1985    0    0    0    1
      1986    0    1    0    0
      1990    0    1    0    0
      1991    0    0    1    0
      1992    1    0    0    0
```

Para exportar un `data.frame` se utiliza la función `write.table(x, file="", sep="", row.names=TRUE, col.names=TRUE)`. El argumento `x` representa el

[8] En un fichero con formato CSV (Comma Separated Value) los datos se separan con comas.

`data.frame`, `file` indica el nombre del fichero destino, `sep` es el carácter separador, `col.names` indica si se debe incluir el nombre de las columnas y `rownames` indica si se debe incluir los nombres de las filas.

```
> write.table(personas, "export-personas-csv.txt", sep=",",
+ row.names=FALSE)
> write.table(personas, file.choose(new=TRUE), sep=",",
+ row.names=FALSE)
```

La función `order(..., decreasing=FALSE)` permite ordenar un `data.frame` por uno o más campos del conjunto de datos.

```
> personas <- read.table("personas.txt")
> personas[order(Sueldo, Nacimiento), ]
   Apellido1 Apellido2 Nombre Nacimiento Sueldo
04    Torres     Ramos  Marta       1992   1000
03   Sánchez Fernández  María       1986   1200
01     López  González   Juan       1990   1200
02     Plata    Suárez   Luis       1991   1300
05      Vega      Ríos  Sofía       1985   1400
> personas[order(Sueldo, decreasing=FALSE), ]
   Apellido1 Apellido2 Nombre Nacimiento Sueldo
04    Torres     Ramos  Marta       1992   1000
01     López  González   Juan       1990   1200
03   Sánchez Fernández  María       1986   1200
02     Plata    Suárez   Luis       1991   1300
05      Vega      Ríos  Sofía       1985   1400
```

Para buscar datos duplicados se utiliza la función `unique(x)`, que devuelve un `data.frame` sin datos duplicados. La función `duplicated(x)` devuelve un vector de valores lógicos que almacena las filas duplicadas.

```
> personas.dup <- read.table("personas-duplicados.txt")
> personas.dup
   Apellido1 Apellido2 Nombre Nacimiento Sueldo
01     López  González   Juan       1990   1200
02     Plata    Suárez   Luis       1991   1300
03     Plata    Suárez   Luis       1991   1300
04   Sánchez Fernández  María       1986   1200
05    Torres     Ramos  Marta       1992   1000
06    Torres     Ramos  Marta       1992   1000
07      Vega      Ríos  Sofía       1985   1400
> unique(personas.dup)
   Apellido1 Apellido2 Nombre Nacimiento Sueldo
01     López  González   Juan       1990   1200
02     Plata    Suárez   Luis       1991   1300
04   Sánchez Fernández  María       1986   1200
05    Torres     Ramos  Marta       1992   1000
07      Vega      Ríos  Sofía       1985   1400
```

Utilizando la función `duplicated(x)` se puede saber cuáles son los registros duplicados del `data.frame`.

```
> personas.dup[duplicated(personas.dup), ]
   Apellido1 Apellido2 Nombre Nacimiento Sueldo
03     Plata    Suárez   Luis       1991   1300
06    Torres     Ramos  Marta       1992   1000
```

La función `merge(x, y)` se utiliza para combinar dos `data.frame`. Esta función permite combinar dos conjuntos de datos que tienen un atributo común. En el siguiente ejemplo, los `data.frame` se combinan utilizando el atributo `id`, que almacena los mismos valores en ambos conjuntos.

```
> d1 <- data.frame(id=letters[1:3], x=10:12)
> d2 <- data.frame(id=letters[1:3], y=20:22)
> d1
  id  x
1  a 10
2  b 11
3  c 12
> d2
  id  y
1  a 20
2  b 21
3  c 22
> merge(d1, d2)
> merge(d1, d2)
  id  x  y
1  a 10 20
2  b 11 21
3  c 12 22
```

Si los valores del atributo `id` no coinciden en los conjuntos `x`, `y`, cuando se ejecuta la función `merge(x, y)` se obtienen solo los valores repetidos en ambos conjuntos o, en su caso, el conjunto vacío.

```
> d1 <- data.frame(id=letters[1:3], x=10:12)
> d3 <- data.frame(id=letters[3:5], y=23:25)
> d3
  id  y
1  c 23
2  d 24
3  e 25
> merge(d1, d3)
  id  x  y
1  c 12 23
```

Para evitar el conjunto vacío, se puede utilizar el atributo `ALL` con el valor `TRUE` para completar con valores `NA` los conjuntos de datos. El argumento `ALL=TRUE` completa los valores no definidos en ambos `data.frame`. Si se desea completar el `data.frame y` con valores `NA`, se debe utilizar el argumento `ALL.x=TRUE`. Para completar los valores de `x`, se debe utilizar `ALL.y=TRUE`.

```
> merge(d2, d3, all.x=TRUE)
  id  y
1  a 20
2  b 21
3  c 22
> merge(d2, d3, all.y=TRUE)
  id  y
1  c 23
2  d 24
3  e 25
```

El argumento by indica el nombre del atributo por el que se realiza la combinación de los dos data.frame. Si ambos tienen el mismo nombre, no es necesario indicarlo expresamente.

```
> d1 <- data.frame(id=letters[1:6], x=10:15)
> d2 <- data.frame(id=letters[1:6], y=20:25)
> merge(d1, d2, by="id")
  id  x  y
1  a 10 20
2  b 11 21
3  c 12 22
4  d 13 23
5  e 14 24
6  f 15 25
```

Si el atributo por el que se desea combinar los data.frame no tiene el mismo nombre, es necesario indicar el atributo correspondiente a cada data.frame.

```
> d1 <- data.frame(id1=letters[1:6], x=10:15)
> d2 <- data.frame(id2=letters[1:6], y=20:25)
> merge(d1, d2, by.x="id1", by.y="id2")
  id1  x  y
1   a 10 20
2   b 11 21
3   c 12 22
4   d 13 23
5   e 14 24
6   f 15 25
```

La función reshape(data, varying = NULL,direction) modifica la organización de los datos de un data.frame y permite pasar de dos o más columnas de datos a una sola. En el siguiente ejemplo, las columnas y.1, y.2 se combinan en una sola aplicando el argumento direction con valor long.

```
> d1 <- data.frame(id=letters[1:3], x=10:12,
+ y.1=letters[1:3], y.2=letters[7:9])
> d1
  id  x y.1 y.2
1  a 10   a   g
2  b 11   b   h
3  c 12   c   i
> reshape(d1, varying=c("y.1", "y.2"), direction="long")
    id  x time y
a.1  a 10    1 a
b.1  b 11    1 b
c.1  c 12    1 c
a.2  a 10    2 g
b.2  b 11    2 h
c.2  c 12    2 i
```

La función match(x, ...) encuentra un valor en un campo de un data.frame. Devuelve un vector indicando la posición donde aparece el valor de búsqueda y NA en cualquier otro caso.

```
> d1 <- data.frame(id=c("a", "b", "c", "d", "e", "f", "g", "f"))
> with(d1, match(id, "d"))
[1] NA NA NA  1 NA NA NA NA
```

La función `complete.cases()` devuelve un vector con valores lógicos que indica con `FALSE` los valores `NA` en un conjunto de datos. La función `na.omit()`elimina los valores `NA` de un conjunto de datos, `na.fail()` devuelve un error cuando existen valores `NA`.

```
> d1 <- c(1, 2, 4, 6, NA, 7, NA)
> complete.cases(d1)
[1]  TRUE  TRUE  TRUE  TRUE FALSE  TRUE FALSE
```

Una vez creado un `dataframe`, se pueden aplicar funciones estadísticas y crear tablas para analizar los datos o hacer gráficos.

```
> setwd("c:/Mis documentos de trabajo/R Data")
> vehiculos <- read.csv("vehiculos.txt")
> vehiculos
   marca modelo         tipo combustible potencia precio
1   Audi     A3 Descapotable    Gasolina      105  29300
2   Audi    TTS        Coupe    Gasolina      272  56000
3   Audi    TTS        Coupe    Gasolina      272  60000
4    BMW     X3  Todoterreno      Diesel      143  37900
5    BMW     X5  Todoterreno      Diesel      254  63100
6   Fiat    500      Turismo    Gasolina      100  16000
7   Fiat  Punto      Turismo    Gasolina       77  13150
8   Ford Fiesta      Turismo    Gasolina       60  13350
9   Ford  Focus      Turismo    Gasolina      115  17150
10 Honda  Civic      Turismo    Gasolina      100  19600
11  Seat  Ibiza      Turismo    Gasolina       85  14850
12  Seat Toledo      Turismo      Diesel      105  19850
13  Seat   León      Turismo      Diesel       90  18400
14    VW   Golf      Turismo    Gasolina      105  18500
15    VW Beetle Descapotable    Gasolina      105  23000
> vehiculos$marca
[1] Audi  Audi  Audi  BMW   BMW   Fiat  Fiat  Ford  Ford  Honda Seat  Seat
[13] Seat  VW    VW
Levels: Audi BMW Fiat Ford Honda Seat VW
```

Uso de un objeto `table`:

```
> table(vehiculos$tipo)
       Coupe Descapotable  Todoterreno      Turismo
           2            2            2            9

> table(vehiculos$tipo)/length(vehiculos$tipo)
       Coupe Descapotable  Todoterreno      Turismo
   0.1333333    0.1333333    0.1333333    0.6000000

> table(vehiculos$tipo)/length(vehiculos$tipo)*100
       Coupe Descapotable  Todoterreno      Turismo
    13.33333     13.33333     13.33333     60.00000

> table(vehiculos$marca, vehiculos$tipo)
        Coupe Descapotable Todoterreno Turismo
  Audi      2            1           0       0
  BMW       0            0           2       0
  Fiat      0            0           0       2
  Ford      0            0           0       2
  Honda     0            0           0       1
  Seat      0            0           0       3
  VW        0            1           0       1
```

```
> vehiculos[order(vehiculos$precio), ]
   marca modelo         tipo combustible potencia precio
7   Fiat  Punto      Turismo    Gasolina       77  13150
8   Ford Fiesta      Turismo    Gasolina       60  13350
11  Seat  Ibiza      Turismo    Gasolina       85  14850
6   Fiat    500      Turismo    Gasolina      100  16000
9   Ford  Focus      Turismo    Gasolina      115  17150
13  Seat   León      Turismo      Diesel       90  18400
14    VW   Golf      Turismo    Gasolina      105  18500
10 Honda  Civic      Turismo    Gasolina      100  19600
12  Seat Toledo      Turismo      Diesel      105  19850
15    VW Beetle Descapotable    Gasolina      105  23000
1   Audi     A3 Descapotable    Gasolina      105  29300
4    BMW     X3  Todoterreno      Diesel      143  37900
2   Audi    TTS        Coupe    Gasolina      272  56000
3   Audi    TTS        Coupe    Gasolina      272  60000
5    BMW     X5  Todoterreno      Diesel      254  63100
```

#### 2.3.5.3. Funciones de uso común

A continuación se describen las funciones de uso común con `data.frame`.

| Función | Descripción |
|---|---|
| setwd(str) | Define el directorio de trabajo |
| getwd() | Devuelve el directorio de trabajo |
| scan() | Permite introducir un datos desde el teclado |
| read.table() | Lee un fichero y lo almacena en un data.frame |
| read.csv() | Lee un fichero en formato CSV y lo almacena en un data.frame |
| read.fwf() | Lee una tabla donde cada columna de datos tiene un ancho fijo |
| write.table() | Escribe el contenido de un data.frame en un fichero |
| write.csv() | Escribe el contenido de un data.frame en un fichero CSV, con los datos separados con comas |
| file.choose() | Selecciona el fichero origen de datos |
| with() | Facilita el uso de fuciones sin el argumento data |
| order() | Devuelve un vector con las posiciones de los elementos ordenados |
| unique() | Elimina los valores duplicados de un conjunto de datos |
| match() | Encuentra un valor en un conjunto de datos |
| merge() | Combina dos data.frame |
| reshape() | Transforma un conjunto de datos a formato long, wide |
| na.ommit() | Elimina los valores NA de un conjunto de datos |
| na.fail() | Devuelve un error cuando existen valores NA |
| complete.cases() | Devuelve un vector con valores lógicos que indica con FALSE los valores NA en un conjunto de datos |
| attach() | Añade el data.frame al espacio de nombres de R |
| detach() | Elimina el data.frame del espacio de nombres de R |

## 2.4. Estructuras de selección

R utiliza las sentencias `if` e `if else` para controlar el flujo de un programa y la función `switch` para seleccionar entre elementos de una lista.

### 2.4.1. if

La sentencia `if` es un caso particular de `if else` que ejecuta condicionalmente una sentencia. Si el valor de la condición es verdadero, entonces se ejecuta `sentencia-1`.

Una sentencia `if` se puede expresar en una sola línea.

```
if (condicion) sentencia-1
```

Si no basta una línea para expresar la sentencia, es necesario utilizar llaves para delimitar el bloque `sentencia-1`.

```
if (condición) {
   sentencia-1
}
```

Para toda sentencia `if`, la condición se evalúa para obtener un valor. Este resultado puede ser un vector de tipo lógico o numérico, en cualquier otro caso se produce un error. Si el resultado es un vector lógico y su primer elemento es `TRUE`, se ejecuta `sentencia-1`. Si el resultado es un vector numérico y su primer elemento es distinto de cero se ejecuta `sentencia-1`.

El siguiente ejemplo, comprueba si el vector `x` de números enteros tiene valores mayores de cero.

```
> x <- c(-2:2)
> if (any(x > 0)) cat("x tiene valores mayores de cero")
x tiene valores mayores de cero
```

### 2.4.2. if else

La sentencia `if else` ejecuta condicionalmente una sentencia entre dos bloques de sentencias. Si el valor de la condición es verdadero, entonces se ejecuta el bloque `sentencia-1`, en caso contrario, se ejecuta el bloque `sentencia-2`. Una sentencia `if else` se puede expresar en una sola línea.

```
if (condicion) sentencia-1 else sentencia-2
```

Si no basta una línea para expresar la sentencia, es necesario utilizar llaves para delimitar el bloque `sentencia-1`. El bloque `sentencia-2` solo debe delimitarse con llaves si está formado por más de una sentencia.

```
if (condición) {
   sentencia-1
} else sentencia-2
```

De forma general, toda sentencia `if else` se puede expresar con la siguiente sintaxis:

```
if (condición) {
   sentencia-1
} else {
   sentencia-2
}
```

Para toda sentencia `if else`, la condición se evalúa para obtener un valor. Este resultado puede ser un vector de tipo lógico o numérico, en cualquier otro caso se produce un error. Si el resultado es un vector lógico y su primer elemento es `TRUE`, se ejecuta el bloque `sentencia-1`, si su valor es `FALSE`, se ejecuta el bloque `sentencia-2`. Si el resultado es un vector numérico y su primer elemento es cero se ejecuta el bloque `sentencia-2`, en cualquier otro caso se ejecuta el bloque `sentencia-1`.

El siguiente ejemplo, comprueba si el vector `x` de números enteros tiene valores mayores de cero.

```
> x <- c(-2:2)
> x
[1] -2 -1 0 1 2
> if (any(x > 0)) y <- TRUE else y <- FALSE
> y
[1] TRUE
```

La misma sentencia, expresada con llaves para delimitar los bloques del caso verdadero y el caso falso del `if else`.

```
> if (any(x > 0)) {
+   y <- TRUE
+ } else x <- FALSE
> y
[1] TRUE
> if (any(x > 0)) {
+   y <- TRUE
+ } else {
+   x <- FALSE
+ }
> y
[1] TRUE
```

En caso de que existan varias cláusulas `if else`, R admite el uso de `if` anidados.

```
if (condición-1) {
   sentencia-1
} else if (condición-2) {
   sentencia-2
} else if (condicion-3) {
   sentencia-3
} else if (condicion-4) {
   sentencia-4
} else sentencia-5
```

El siguiente `if else` comprueba si un vector tiene elementos positivos, iguales a cero o negativos, considerando que solo se puede cumplir una condición a la vez.

```
> x <- c(1:10)
> if (any(x > 0)) {
+ cat("x tiene valores mayores de cero")
+ } else if (any(x == 0)) {
+ cat("x tiene valores iguales a cero")
+ } else cat("x tiene valores menores de cero")
x tiene valores mayores de cero
```

Una sentencia `if else` se puede utilizar para inicializar el valor de un objeto.

```
z <- if (any(x > 0)) TRUE else FALSE
```

### 2.4.3. switch

El `switch` es una estructura de control de flujo como la de muchos otros lenguajes de programación. Desde un punto de vista técnico, el `switch` de R es una función.

La sintaxis de la función `switch()` es:

```
switch(sentencia, lista-de-valores)
```

La función `switch()` calcula el valor de la sentencia. Si el valor obtenido es un número entre 1 y el total de elementos de la lista, entonces devuelve el elemento almacenado en esa posición de la lista.

```
> switch(3, "a", "e", "i", "o", "u")
[1] "i"
> switch((3+2)/2, "a", "e", "i", "o", "u")
[1] "e"
> switch(10, "a", "e", "i", "o", "u")
>
```

En este ejemplo, el primer `switch()` devuelve el elemento 3 de la lista de vocales. El segundo devuelve el elemento 2 de la lista y el último devuelve el valor NULL.

La función `switch()` devuelve el elemento de la lista que corresponda con el valor que resulta de evaluar la sentencia de la función, independientemente de que se trate de un valor, un vector o una lista. Si la lista de valores tiene nombres, entonces el valor de la sentencia del `switch()` debe coincidir con uno de los nombres de la lista.

```
> switch("e", a="A", e="E", i="I", o="O", u="U")
[1] "E"
```

## 2.5. Estructuras de repetición

Las estructuras de repetición permiten repetir muchas veces un bloque de sentencias. A estas estructuras también se les conoce como estructuras iterativas o bucles.

Como las estructuras de selección, las estructuras de repetición se pueden combinar y anidar. Es frecuente utilizar una estructura de repetición que contenga un bloque de sentencias que combine otras estructuras de repetición y de selección.

### 2.5.1. while

La estructura de repetición `while` repite el bloque de sentencias mientras la condición es verdadera.

Un `while` se puede expresar en una sola línea.

```
while (condición) sentencia
```

Si no basta una línea para expresar la sentencia, es necesario utilizar llaves para delimitar el bloque de sentencias.

```
while (condición) {
   sentencia
}
```

Cuando se ejecuta un `while`, lo primero que hace es evaluar la condición. Si es verdadera ejecuta el bloque de sentencias, si es falsa finaliza el `while`. En cada iteración, cuando finaliza la ejecución del bloque de sentencias se vuelve a evaluar la condición. De nuevo, si es verdadera ejecuta una vez más el bloque de sentencias, si es falsa finaliza el `while`. Cuando esto se produce, el flujo del programa continúa en la sentencia inmediatamente posterior al `while`. Si la condición siempre es verdadera, entonces el `while` nunca termina y se ejecuta indefinidamente. Esto se conoce como bucle infinito.

```
> x <- c(1:10)
> k <- 1
> while(k <= 10) {
+   print(x[k])
+   k <- k + 2
+ }
[1] 1
[1] 3
[1] 7
[1] 7
[1] 9
```

### 2.5.2. repeat

La estructura de repetición `repeat` ejecuta el bloque de sentencias indefinidamente hasta que se ejecuta un `break`.

Un `repeat` se puede expresar en una sola línea.

```
repeat sentencia
```

Si no basta una línea para expresar la sentencia, es necesario utilizar llaves para delimitar el bloque de sentencias.

```
repeat {
   sentencia
}
```

El bloque de sentencias de un `repeat` debe incluir una condición de fin utilizando un `break`.

```
> x <- c(1:10)
> k <- 1
> repeat {
+    print(x[k])
+    k <- k + 1
+    if (k > 5) break
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

### 2.5.3. for

La estructura `for` repite el bloque de sentencias para todos los elementos del vector.

Un `for` se puede expresar en una sola línea.

```
for (objeto en vector) sentencia
```

Si no basta una línea para expresar la sentencia, es necesario utilizar llaves para delimitar el bloque de sentencias.

```
for (objeto en vector) {
   sentencia
}
```

El `for` se repite para cada uno de los elementos del vector o de la lista indicada.

```
> x <- c(1:10)
for(i in 1:3) print(x[i])
[1] 1
[1] 2
[1] 3
```

Uso de un `for` con una lista de caracteres.

```
> v <- list("a", "e", "i", "o", "u")
> for(i in 1:3) print(v[i])
[[1]]
[1] "a"
[[1]]
[1] "e"
[[1]]
[1] "i"
```

El `for` también se puede expresar utilizando la función `seq_along()`.

```
> for(i in seq_along(v)) print(v[i])
```

## 2.6. Funciones

El desarrollo de funciones personalizadas por el usuario es una característica muy importante de R porque incrementa su capacidad y potencia. R es un lenguaje de programación funcional, de ahí que las funciones se puedan utilizar como argumentos de funciones y también como valores de retorno de funciones.

### 2.6.1. Declaración de funciones

Un objeto de tipo función se define por una expresión de asignación de la forma:

```
function-id <- function(argument-list) {
  body
}
```

Esta expresión define el identificador de la función, la lista de argumentos formales y el cuerpo de la función. La palabra reservada `function` indica a R que es la declaración una función. Los elementos de la lista de argumentos formales de la función se separan con comas. Un argumento formal puede ser cualquier tipo de objeto, incluida una función y el argumento especial '`...`', que se utiliza recibir un número indefinido de argumentos. La lista de argumentos puede incluir valores por defecto, por ejemplo: `name1, name2=default-value2, name3=default-value3`. Es importante señalar que cuando se define una nueva función, si ya existe una función con el mismo nombre, la nueva función sobreescribe a la anterior.

El cuerpo de la función puede ser cualquier expresión válida de R. Normalmente, el cuerpo se delimita por los símbolos '`{`' y '`}`'.

Para ejecutar una función se utiliza una expresión como `id(expr1, expr2, ...)`, donde `id` es el identificador de la función y `expr1`, `expr2`, `...` sus argumentos.

Por ejemplo, se puede definir la función `Echo()` para mostrar datos por la consola.

```
> Echo <- function(x) print(x)
> Echo("Hola mundo")
[1] "Hola mundo"
> print("Hola mundo")
[1] "Hola mundo"
```

La siguiente función calcula la varianza de un conjunto de datos con distribución normal generado de forma aleatoria.

```
VarNormal <- function(n) {
  s <- rnorm(n)
  v <- var(s)
  return(v)
}
```

Para ejecutar una función, basta con indicar su identificador y la lista de argumentos definida:

```
> VarNormal(50)
[1] 0.9811938
```

Si una función tiene un valor de retorno, éste debe devolverse utilizando la función `return()` o `invisible()`. Si se utiliza la función `return()`, la función imprime el resultado y devuelve el valor, si se utiliza `invisible()` solo devuelve el resultado.

```
VarNormal <- function(n) {
  s <- rnorm(n)
  v <- var(s)
  invisble(v)
}
```

El valor de retorno de una función se puede asignar a una variable. En este caso, independientemente de que la función utilice `return()` o `invisible()` para devolver el valor, no se muestra el resultado.

```
> x <- VarNormal(50)
```

### 2.6.2. Argumentos y valores por defecto

Los argumentos formales de una función definen las variables que recibe una función cuando es invocada. Cuando se ejecuta una función, la lista de argumentos se debe expresar en el orden en el que han sido definidos si no se utiliza el nombre del identificador, o en un orden distinto, siempre que se indique el identificador de cada argumento. Por ejemplo, la función `graph()` declara la lista de argumentos `data, caption, color`.

```
Graph <- function(data, caption, boldface) {

}
```

Esta función se puede llamar respetando el orden natural de la declaración de los argumentos o utilizando expresamente su identificador antes de indicar el valor del argumento. Las siguientes expresiones son válidas y se pueden utilizar indistintamente.

```
> datos <- c(100, 110, 115, 105, 115, 120)
> Graph(datos, "Evolucion de ventas", TRUE)
> Graph(datos, boldface=TRUE, caption="Evolucion de ventas")
> Graph(caption="Evolucion de ventas", boldface=TRUE, data=datos)
```

En la declaración de una función, R permite especificar valores por defecto para sus argumentos. En estos casos, cuando un argumento tiene definido un valor por defecto, se puede omitir al llamar a la función.

```
Graph <- function(data, caption, boldface=FALSE) {
}
```

En esta nueva declaración de la función `graph()`, el valor por defecto del argumento `boldface` es `FALSE` y puede omitirse cuando se llama a la función.

```
> Graph(datos, "Evolucion de ventas")
> Graph(datos, "Evolucion de ventas", FALSE)
```

En este ejemplo, ambas llamadas son equivalentes y se obtiene el mismo resultado. Si se desea llamar a la función utilizando un dato distinto del valor por defecto de un argumento, entonces se debe indicar de forma expresa.

```
> Graph(datos, "Evolucion de ventas", TRUE)
```

La siguiente función realiza una búsqueda binaria en un vector ordenado de números.

```
BusquedaBinaria <- function(x, n) {
  inf <- 1
  sup <- length(x)
  res <- FALSE
  while (inf <= sup) {
    med <- (inf + sup)/2
    if (x[med] == n) {
      res <- TRUE
      break
    }
    else if (x[med] > n) {
      sup <- med - 1
    } else {
      inf <- med + 1
    }
  }
  return(res)
}

> numeros <- c(1, 2, 3, 4, 6, 7, 8, 9, 10, 15, 17, 20, 45, 51, 60, 68, 75)
> BusquedaBinaria(numeros, 3)
[1] TRUE
> BusquedaBinaria(numeros, 5)
[1] FALSE
```

### 2.6.3. El argumento '…'

Existe un tipo de argumento especial que se declara expresamente '`...`'. Normalmente se utiliza para pasar una lista indeterminada de argumentos a funciones genéricas de R, como `print()` o `plot()`.

### 2.6.4. Evaluación de funciones

Cuando se ejecuta una función, R crea un registro de activación[9] y compara uno a uno los argumentos formales definidos en la declaración con los argumentos que recibe la función. Antes de evaluar una función, se comparan los argumentos formales declarados en la función con los argumentos actuales que recibe en tiempo de ejecución.

1. Comparación exacta de las etiquetas de los argumentos. Compara las etiquetas de los argumentos formales con las etiquetas de los argumentos actuales. Si un argumento actual coincide con varios argumentos formales, se produce un error.
2. Comparación parcial de las etiquetas de los argumentos. Con el resto de argumentos para los que aún no se ha encontrado una correspondencia, se aplica la comparación parcial. Si el nombre de un argumento actual coincide exactamente con la primera parte de un argumento formal, entonces ambos argumentos se comparan.
3. Comparación posicional de los argumentos. Los argumentos actuales que aún no tienen correspondencia se consideran argumentos sin identificador, en su orden natural. Si se ha definido el argumento '. . .', éste recoge el resto de argumentos.

Si después de aplicar estas reglas descritas hay argumentos actuales que aún no tienen asociado un argumento formal, entonces se produce un error.

Los argumentos se pasan por valor y por tanto se tratan como variables locales que se inicializan con el valor asignado al argumento actual. Si se modifica el valor de un argumento actual dentro de una función, no afecta al valor de la variable en el registro de activación que ha invocado a la función. R utiliza la evaluación perezosa para los argumentos de una función. Esto significa que los argumentos no se evalúan hasta que es necesario, lo que en ciertos casos, nunca sucede.

## 2.7. Ámbito léxico

Las asignaciones que se realizan dentro del cuerpo de una función son locales al ámbito de ejecución de la función. Esto significa que las asignaciones son temporales y que estos valores se pierden cuando finaliza la ejecución de la función.

[9] Un registro de activación o 'frame' es el espacio de memoria que almacena toda la información sobre el estado actual de la ejecución de R, antes de invocar a otra función

## 2.8. Excepciones

El manejo de excepciones de R se basa en el uso de las funciones `stop` y `warning`. La variable `warn` permite modificar el comportamiento de la función `warning`.

### 2.8.1. Funciones de manejo de excepciones

La función `stop` imprime el mensaje que recibe como argumento y detiene la ejecución del programa.

La función `warning` recibe una cadena de caracteres como argumento. Se utiliza para imprimir los mensajes correspondientes a los 'warnings' que se producen durante la ejecución de un programa. El comportamiento de la función `warning` es diferente según el valor de la variable `warn`. Si `warn` es menor de 0, se ignoran los 'warnings'; si `warn` es 0, se crea la variable `last.warning`, un vector que almacena los mensajes asociados a cada llamada a la función `warning`; si `warn` es 1, se imprimen los mensajes; finalmente, si `warn` es mayor de 1, los 'warnings' se tratan como errores.

```
CovarianzaC <- function(x, y) {
  n <- length(x)

  if (n <= 1)
    stop("¡El número de elementos del vector x debe ser mayor de 1!")
  else
    if (n != length(y))
      stop("¡Los vectores x, y deben tener el mismo númeo de elementos!")

      if (TRUE %in% is.na(x) || TRUE %in% is.na(y)) {
        stop("!Los vectores x, y no pueden tener valores NA!")
      }

  if (!is.loaded("ccov"))
    dyn.load("LibC.dll")

  out <- .C("ccov", x=as.double(x), y=as.double(y), n=as.integer(length(x)),
                    sxy=as.double(1))

  return(out$sxy)
}
```

### 2.8.2. Opciones de control de errores

Las variables de control de las excepciones permiten modificar el comportamiento de las funciones de manejo de errores. La variable `warn` se utiliza para controlar la impresión de 'warnings'. La varable `warning.expression` define la expresión que se debe evaluar cuando se produce un 'warning'. Cuando se activa esta variable, se suprimen los mensajes de 'warning'. Por último, la variable `error` define la expresión que se debe evaluar cuando se produce un error de ejecución.

## 2.9. Importación y exportación de datos

Los conjuntos de datos grandes normalmente se leen de ficheros externos. R ofrece funciones de lectura de datos estructurados en forma de tabla.

### 2.9.1. La función read.table()

La función `read.table()` se utiliza para leer un `data.frame` siempre que los datos de entrada cumplan las siguientes condiciones.

- La primera fila de del archivo debe indicar el nombre de cada columna del `data frame`.
- Cada línea del archivo incluye el identificador de la fila y los valores correspondientes a cada variable de la tabla.

Por defecto, los datos numéricos se leen como variables numéricas y las variables no numéricas como factores.

El fichero '`personas1.txt`' incluye una fila de encabezados de columna y una columna con los identificadores de fila.

```
      Apellido1      Apellido2      Nombre      Nac.        Sueldo
01    López          González       Juan        1990        1.200
02    Plata          Suárez         Luis        1991        1.300
03    Sánchez        Fernández      María       1986        1.200
04    Torres         Ramos          Marta       1992        1.000
05    Vega           Ríos           Sofía       1985        1.400
```

El objeto '`personas`' se inicializa utilizando la función `read.table()`. El fichero '`personas1.txt`' incluye encabezados de columna e identificadores de fila, por lo que no es necesario indicar más argumentos a la función.

Antes de acceder a los ficheros de datos, se modifica el directorio de trabajo de R.

```
> setwd("c:/Mis documentos de trabajo/R Data")
```

La función `read.table()` lee los datos del fichero.

```
> personas <- read.table("personas1.txt")
> personas
   Apellido1 Apellido2 Nombre Nacimiento Sueldo
01     López  González   Juan       1990   1200
02     Plata    Suárez   Luis       1991   1300
03   Sánchez Fernández  María       1986   1200
04    Torres     Ramos  Marta       1992   1000
05      Vega      Ríos  Sofía       1985   1400
```

El fichero ‘personas2.txt’ incluye una fila de encabezados de columna y no tiene identificadores de fila.

```
Apellido1         Apellido2         Nombre         Nac.           Sueldo
López             González          Juan           1990           1.200
Plata             Suárez            Luis           1991           1.300
Sánchez           Fernández         María          1986           1.200
Torres            Ramos             Marta          1992           1.000
Vega              Ríos              Sofía          1985           1.400
```

El objeto personas se inicializa utilizando la función read.table(), utilizando el argumento header=TRUE para indicar que el fichero de datos incluye una fila de encabezado con los nombres de los campos de datos.

```
> personas <- read.table("personas2.txt", header=TRUE)
> personas
  Apellido1 Apellido2 Nombre Nacimiento Sueldo
1     López  González   Juan       1990   1200
2     Plata    Suárez   Luis       1991   1300
3   Sánchez Fernández  María       1986   1200
4    Torres     Ramos  Marta       1992   1000
5      Vega      Ríos  Sofía       1985   1400
```

Si se lee el fichero ‘personas2.txt’ sin utilizar el argumento header con valor TRUE, entonces el objeto personas se inicializa de la siguiente forma.

```
> personas <- read.table("personas2.txt")
> personas
         V1        V2     V3         V4     V5
1 Apellido1 Apellido2 Nombre Nacimiento Sueldo
2     López  González   Juan       1990   1200
3     Plata    Suárez   Luis       1991   1300
4   Sánchez Fernández  María       1986   1200
5    Torres     Ramos  Marta       1992   1000
6      Vega      Ríos  Sofía       1985   1400
```

En este ejemplo, R asigna un valor arbitrario a los nombres de las columnas de datos y considera que la primera fila también es parte de los datos de la tabla. R numera secuencialmente las filas de datos del fichero. El fichero ‘personas3.txt’ no incluye fila de encabezados de columna ni identificadores de fila.

```
López             González          Juan           1990           1.200
Plata             Suárez            Luis           1991           1.300
Sánchez           Fernández         María          1986           1.200
Torres            Ramos             Marta          1992           1.000
Vega              Ríos              Sofía          1985           1.400
```

El objeto personas se inicializa utilizando la función read.table(), sin más argumentos.

```
> personas <- read.table("personas3.txt")
> personas
       V1        V2    V3   V4   V5
1   López  González  Juan 1990 1200
2   Plata    Suárez  Luis 1991 1300
3 Sánchez Fernández María 1986 1200
4  Torres     Ramos Marta 1992 1000
5    Vega      Ríos Sofía 1985 1400
```

De forma similar al ejemplo anterior, R asigna un valor arbitrario a los nombres de las columnas de datos y numera secuencialmente las filas de datos.

### 2.9.2. La función scan()

La función `scan()` es útil para leer vectores de datos del mismo tamaño almacenados en un fichero de datos. Si se utiliza de nuevo el fichero '`personas3.txt`', es necesario indicar que las primeras tres columnas de datos almacenan cadenas de caracteres y las dos últimas valores numéricos. Para ello se utiliza un argumento de tipo lista que representa la estructura y los tipos de datos de los vectores que se van a leer.

```
> datos <- scan("personas3.txt", list("", "", "", 0, 0))
Read 5 records
> datos
[[1]]
[1] "López" "Plata" "Sánchez" "Torres" "Vega"
[[2]]
[1] "González" "Suárez" "Fernández" "Ramos" "Ríos"
[[3]]
[1] "Juan" "Luis" "María" "Marta" "Sofía"
[[4]]
[1] 1990 1991 1986 1992 1985
[[5]]
[1] 1200 1300 1200 1000 1400
```

Para almacenar los datos en distintos vectores basta con asignar el contenido a distintos objetos:

```
> apellido1 <- datos[[1]]
> apellido2 <- datos[[2]]
> nombre <- datos[[3]]
> nacimiento <- datos[[4]]
> sueldo <- datos[[5]]
> nombre
[1] "Juan" "Luis" "María" "Marta" "Sofía"
```

Si se desea asignar un nombre a las columnas de datos del objeto, entonces el objeto lista debe indicar la etiqueta de cada columna de datos además de su tipo.

```
> datos <- scan("personas3.txt",
+            list(apellido1="", apellido2="", nombre="",
+            nacimiento=0, sueldo=0))
Read 5 records
> datos
$apellido1
[1] "López" "Plata" "Sánchez" "Torres" "Vega"
$apellido2
[1] "González" "Suárez" "Fernández" "Ramos" "Ríos"
$nombre
[1] "Juan" "Luis" "María" "Marta" "Sofía"
$nacimiento
[1] 1990 1991 1986 1992 1985
$sueldo
[1] 1200 1300 1200 1000 1400
```

### 2.9.3. La función read.csv()

La función `read.csv()` lee un fichero en formato `CSV` y devuelve un objeto `data.frame`.

```
> vehiculos <- read.csv("vehiculos.txt")
```

### 2.9.4. La función write.csv()

La función `write.csv()` exporta `data.frame` a un fichero con formato `CSV`.

```
> write.csv(vehiculos, "vehiculos.txt")
```

### 2.9.5. La función sink()

La función `sink(file=NULL, append=FALSE)` captura la salida por la consola a un fichero. El argumento `file` indica el nombre del fichero, `append` determina si el fichero se inicializa o no.

```
> setwd("c:/Mis documentos de trabajo/R Data")
> personas <- read.table("personas.txt")
> sink("salida.txt")
> with(personas, table(Nacimiento, Sueldo))
> sink()
```

La salida por la consola se almacena en el fichero '`salida.txt`'.

```
           Sueldo
Nacimiento 1000 1200 1300 1400
      1985    0    0    0    1
      1986    0    1    0    0
      1990    0    1    0    0
      1991    0    0    1    0
      1992    1    0    0    0
```

Para añadir al fichero '`salida.txt`' el resultado de nuevos cálculos sobre el `data.frame`, se debe utilizar el argumento `append=TRUE`.

```
> personas <- read.table("personas.txt", stringsAsFactors=FALSE)
> sink("salida.txt", append=TRUE)
> str(personas)
> sink()
```

El contenido del archivo `salida.txt`.

```
           Sueldo
Nacimiento 1000 1200 1300 1400
      1985    0    0    0    1
      1986    0    1    0    0
      1990    0    1    0    0
      1991    0    0    1    0
      1992    1    0    0    0
'data.frame':    5 obs. of  5 variables:
 $ Apellido1 : chr  "López" "Plata" "Sánchez" "Torres" ...
 $ Apellido2 : chr  "González" "Suárez" "Fernández" "Ramos" ...
 $ Nombre    : chr  "Juan" "Luis" "María" "Marta" ...
 $ Nacimiento: int  1990 1991 1986 1992 1985
 $ Sueldo    : int  1200 1300 1200 1000 1400
```

### 2.9.6. La función capture.output()

La función `capture.output(..., file=NULL, append=false)` permite capturar la salida de la consola y enviarla a una cadena de caracteres o a un fichero. El argumento '`...`' permite llamar a una función, `file` indica el nombre del fichero, `append` determina si el fichero se inicializa o no.

```
> setwd("c:/Mis documentos de trabajo/R Data")
> personas <- read.table("personas.txt")
> capture.output(with(personas, table(Nacimiento, Sueldo)), file="out.txt")
```

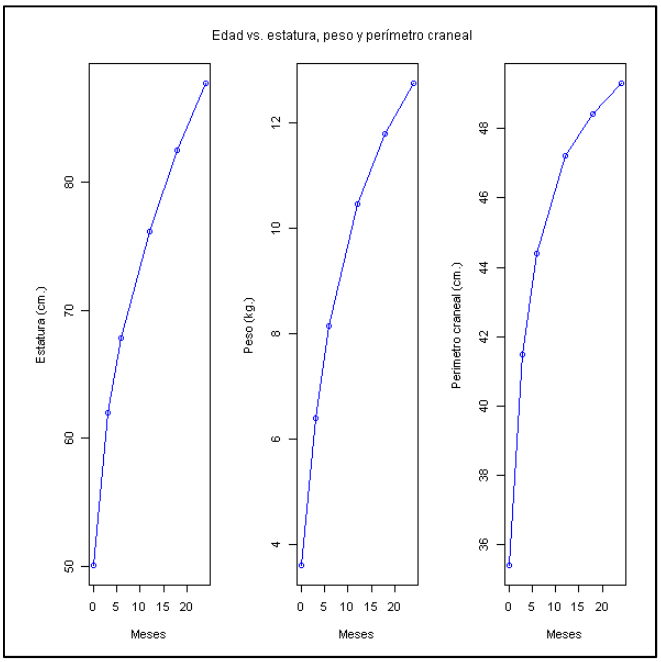
Para añadir al fichero '`salida.txt`' el resultado de nuevos cálculos sobre el `data.frame`, se debe utilizar el argumento `append=TRUE`.

```
> personas <- read.table("personas.txt", stringsAsFactors=FALSE)
> capture.output(str(personas), file="salida.txt", append=TRUE)
```

## 2.10. Gráficos

R ofrece funciones para realizar gráficos de calidad. Normalmente es necesario combinar funciones de alto nivel como `plot()` o `boxplot()` con funciones de bajo nivel como `text()`, `mtext()` o `legend()` para personalizar un gráfico.

```
> meses <- c(0, 3, 6, 12, 18, 24)
> estatura <- c(50.10, 62.00, 67.85, 76.15, 82.40, 87.65)
> peso <- c(3.60, 6.40, 8.15, 10.45, 11.80, 12.75)
> perimetro.craneal.niño <- c(35.40, 41.50, 44.40, 47.20, 48.40, 49.30)
> par(mfrow=c(1,3), oma=c(0, 2, 0, 0))
> plot(x=meses, y=estatura, type="o", col="blue",
+      xlab="Meses", ylab="Estatura (cm.)")
> plot(x=meses, y=peso, type="o", col="blue",
+      xlab="Meses", ylab="Peso (kg.)")
> plot(x=meses, y=perimetro.craneal.niño, type="o", col="blue",
+      xlab="Meses", ylab="Perimetro craneal (cm.)")
> mtext("Edad vs. estatura y peso", side=3, outer=TRUE, line=-2.5, cex=0.75)
```




A continuación se describen las funciones de uso común para gráficos.

| Función | Descripción |
|---|---|
| par() | Define los parámetros del gráfico |
| plot() | Gráfico de líneas |
| hist() | Histograma |
| boxplot() | Gráfico de cajas |
| text() | Imprime un texto en el área del gráfico |
| mtext() | Imprime un texto fuera del área del gráfico |
| title() | Define el título del gráfico |
| legend() | Define la leyenda del gráfico |

# Capítulo 3. Programación orientada a objetos

La programación orientada a objetos es un paradigma de programación que se define por el uso de clases y objetos. Una clase es una representación abstracta que define las características comunes de un conjunto de objetos. Un objeto es una instancia de una clase, tiene una identidad propia y un estado. La identidad de un objeto se define por su identificador y su estado por el valor de sus atributos. Los métodos son funciones que se definen dentro del ámbito de una clase y realizan cálculos y operaciones con los objetos de la clase, es decir, definen el comportamiento de los objetos. El modelo de programación orientada a objetos facilita la encapsulación del código, su reutilización y el mantenimiento del software. De ahí que su aplicación es necesaria para desarrollar paquetes en R.

R hereda los sistemas de clases S3 y S4 del lenguaje S y permite desarrollar programas utilizando cualquiera de estos sistemas [6-9, 12]. Las clases S3 son simples, flexibles y se utilizan en muchos paquetes de R. Las clases S4, en cambio, son más formales y rigurosas, garantizan que cualquier objeto cumple todos los requisitos especificados en la clase, pero su uso está menos extendido. Bioconductor[10], un paquete diseñado para analizar datos genómicos se ha desarrollado utilizando clases y métodos S4. El aspecto formal del sistema de clases S4 facilita la coordinación y la colaboración de las aportaciones de las personas que trabajan simultáneamente en un proyecto complejo como éste.

La programación orientada a objetos en R se basa en los sistemas de clases S3 y S4, el uso de 'funciones genéricas' y un sistema de 'dispatching' de métodos. Una 'función genérica' es una función que tiene asociados uno o más métodos definidos por el usuario. El sistema de 'dispatching' determina cuál de los métodos definidos para una 'función genérica' se debe ejecutar para cada objeto, considerando los argumentos que recibe la función cuando es invocada. Los sistemas de clases S3 y S4 permiten definir nuevos métodos para modificar el comportamiento estándar de los métodos de R. Esta característica de la programación orientada a objetos se denomina sobreescritura de métodos. Consiste en el uso de nuevos métodos que modifican el comportamiento estándar de métodos como `print`, `summary` o `plot` que se aplican a todos los objetos de R. Para personalizar estos métodos basta con definir una nueva clase y los nuevos métodos, según las características y las necesidades específicas de la clase.

[10] http://www.bioconductor.org/

Para mostrar el uso de clases S3 y S4 en R, se utilizan datos sobre la evolución del crecimiento en niños y niñas, descargados de la web del 'Centers for Disease Control and Prevention'[11] (CDC). La base de datos almacena la edad, expresada en meses, la estatura, el peso y el diámetro craneal de cada niño, como se muestra a continuación.

| Edad (meses) | Estatura (cm.) | Peso (kg.) | Perimetro craneal (cm.) |
|---|---|---|---|
| 0 | 50.10 | 3.60 | 35.40 |
| 3 | 62.00 | 6.40 | 41.50 |
| 6 | 67.85 | 8.15 | 44.40 |
| 12 | 76.15 | 10.45 | 47.20 |
| 18 | 82.40 | 11.80 | 48.40 |
| 24 | 87.65 | 12.75 | 49.30 |

La clase de ejemplo se denomina `RegistroInfantil` y se compone de los atributos `id` y `datos`. El atributo `datos` es un `data.frame` que almacena los valores `mes`, `estatura`, `peso` y `perimetro.craneal`.

## 3.1. Clases S3

El atributo `class` asocia una clase S3 a un objeto de R. Este atributo es un vector de caracteres que almacena la clase de la que hereda un objeto. Para saber la clase a la que pertenece un objeto se utiliza la función `class()`.

### 3.1.1. Declaración de clases

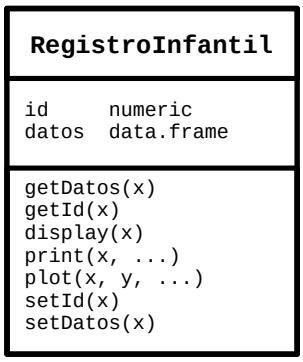
Par declarar una clase S3 se define una función que inicialice y devuelva un objeto de la clase. La función `RegistroInfantil` recibe como argumentos un valor numérico `id` que representa el identificador del objeto, un vector que almacena la edad y tres vectores que almacenan la estatura, el peso y el perímetro craneal para las seis medidas tomadas desde el nacimiento niño hasta que alcanza los 24 meses de edad. El objeto que devuelve la función es una lista compuesta por dos elementos: `id` es el identificador del registro y `datos` un `data.frame` que almacena la tabla de datos que se construye con los vectores `mes`, `estatura`, `peso` y `perimetro.craneal`.

Para definir el objeto de la clase `RegistroInfantil` se utilizan los siguientes datos:

```
> meses <- c(0, 3, 6, 12, 18, 24)
> peso <- c(3.60, 6.40, 8.15, 10.45, 11.80, 12.75)
> estatura <- c(50.10, 62.00, 67.85, 76.15, 82.40, 87.65)
> perimetro.craneal <- c(35.40, 41.50, 44.40, 47.20, 48.40, 49.30)
```

[11] http://www.cdc.gov/

El diagrama de clases muestra los atributos y los métodos de la clase.




La declaración de la clase `RegistroInfantil`.

```
RegistroInfantil <- function(id, mes, estatura, peso, perimetro.craneal) {
  x <- list(id=id,
            datos=data.frame(mes=mes, estatura=estatura, peso=peso,
                             perimetro.craneal=perimetro.craneal))

  class(x) <- "RegistroInfantil"

  return(x)
}
```

Para crear un objeto de la clase se ejecuta la función `RegistroInfantil`.

```
> obj <- RegistroInfantil(1, meses, estatura, peso, perimetro.craneal)
```

La función `class` indica la clase a la que pertenece el objeto.

```
> class(obj)
[1] "RegistroInfantil"
```

### 3.1.2. Declaración de métodos

Si se ejecuta el método estándar `print`, R muestra los atributos, los valores almacenados en el objeto y la clase a la que pertenece.

```
> print(obj)
$id
[1] 1

$datos
  mes estatura.cm peso.kg perimetro.craneal.cm
1   0       50.10    3.60                 35.4
2   3       62.00    6.40                 41.5
3   6       67.85    8.15                 44.4
4  12       76.15   10.45                 47.2
5  18       82.40   11.80                 48.4
6  24       87.65   12.75                 49.3

attr(,"class")
[1] "RegistroInfantil"
```

Para modificar el comportamiento estándar de un método de R para los objetos de una clase S3, se debe definir un nuevo método cuya declaración coincida con la declaración del método de R. Por ejemplo, para ver la declaración del método `print` de R basta con introducir su identificador en la consola.

```
> print
function (x, ...)
```

La declaración del método `print` es `function(x, ...)`. El nuevo método `print` debe utilizar esta misma declaración de argumentos. Esto es muy importante, ya que evita errores durante el proceso de chequeo de los métodos S3 de un paquete.

Para definir un método `print` específico para la clase `RegistroInfantil`, se utiliza el nombre del método estándar seguido de un punto y el nombre de la clase. En este ejemplo, el identificador del nuevo método es `print.RegistroInfantil`.

```
print.RegistroInfantil <- function(x, ...) {
  cat("id:", x$id, "\n")

  print(x$datos)
}
```

Este método sobreescribe el método `print` de R y la salida es:

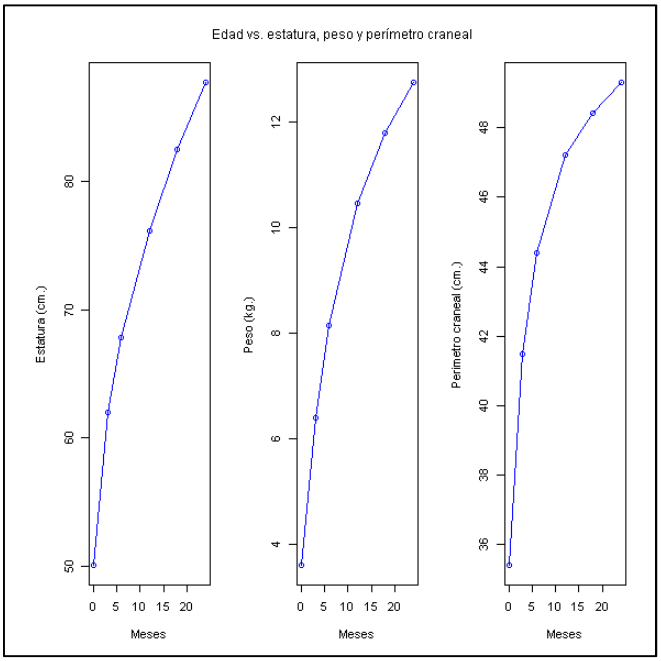
```
> print(obj)
id: 1
  mes estatura.cm peso.kg perimetro.craneal.cm
1   0       50.10    3.60                35.4
2   3       62.00    6.40                41.5
3   6       67.85    8.15                44.4
4  12       76.15   10.45                47.2
5  18       82.40   11.80                48.4
6  24       87.65   12.75                49.3
```

Para definir un método `plot` para la clase `RegistroInfantil` se utiliza el identificador `plot.RegistroInfantil` y la declaración `function(x, y, ...)`, que coincide con el método `plot` de R. Este método muestra tres gráficos para ver la evolución de la estatura, el peso y el diámetro craneal del niño durante sus dos primeros años de vida.

```
plot.RegistroInfantil <- function(x, y, ...) {
  datos <- x$datos

  par(mfrow=c(1,3), oma=c(0, 2, 0, 0))

  plot(datos$mes, datos$estatura, type="o", col="blue",
       xlab="Meses", ylab="Estatura (cm.)")
  plot(datos$mes, datos$peso, type="o", col="blue",
       xlab="Meses", ylab="Peso (kg.)")
  plot(datos$mes, datos$perimetro.craneal, type="o", col="blue",
       xlab="Meses", ylab="Perimetro craneal (cm.)")

  mtext("Edad vs. estatura, peso y perímetro craneal", side=3,
        outer=TRUE, line=-2.5, cex=0.75)
}
```

El método `plot` construye tres gráficos que muestran la evolución de la estatura, el peso y el diámetro craneal con respecto a la edad del niño.

```
> plot(obj)
```




Para definir un método totalmente nuevo que no sobreescriba un método de R, se utiliza la función `UseMethod()` para declarar una 'función genérica' y después se declara el nuevo método. El identificador del nuevo método es el nombre del método seguido de un punto y el nombre de la clase. Por ejemplo, para definir un método `display` para la clase `RegistroInfantil`, se utiliza el identificador `display.RegistroInfantil`.

```
display <- function (x) UseMethod("display", x)

display.RegistroInfantil <- function(x) {
  print(x$datos)
}
```

Este método recibe un objeto en el argumento `x`. Es similar al método `print`, pero a diferencia de éste, no muestra el identificador del objeto.

```
> display(obj)
  mes estatura.cm peso.kg perimetro.craneal.cm
1   0       50.10    3.60                 35.4
2   3       62.00    6.40                 41.5
3   6       67.85    8.15                 44.4
4  12       76.15   10.45                 47.2
5  18       82.40   11.80                 48.4
6  24       87.65   12.75                 49.3
```

El uso de métodos genéricos permite definir métodos 'get' y 'set' para los atributos de la clase. Los métodos 'get' extraen los valores de los atributos y los métodos 'set' modifican los valores almacenados en un objeto

El método `getId` extrae el valor del atributo `id`.

```
getId <- function (x) UseMethod("getId", x)

getId.RegistroInfantil <- function(x) {
  return(x$id)
}
```

El método `getDatos` extrae el valor del atributo `datos`.

```
getDatos <- function (x) UseMethod("getDatos", x)

getDatos.RegistroInfantil <- function(x) {
  return(x$datos)
}
```

Para definir un método 'set' también se utiliza la función `UseMethod()`, pero, a diferencia de los métodos 'get', el identificador debe incluir el operador de asignación '`<-`' al final. Por ejemplo, el método `setId` se denomina "`setId<-`" y su declaración es `function(x, value)`, que incluye el valor que se asigna al objeto.

El método "`setId<-`" modifica el valor del atributo `id`.

```
"setId<-" <- function (x, value) UseMethod("setId<-", x)

"setId<-.RegistroInfantil" <- function(x, value) {
  x$id <- value
  return(x)
}
```

El método "`setDatos<-`" modifica el valor del atributo `datos`.

```
"setDatos<-" <- function (x, value) UseMethod("setDatos<-", x)

"setDatos<-.RegistroInfantil" <- function(x, value) {
  x$datos <- value
  return(x)
}
```

Si se desea, se puede definir un método 'get' para sobrecargar el operador '`[`' y acceder a los atributos de la clase por su nombre. El identificador del método se obtiene concatenando el operador '`[`'seguido de un punto y el nombre de la clase. En este ejemplo, el identificador es "`[.RegistroInfantil`" y la declaración de los argumentos es `function(x, i, j, drop)` para acceder a los elementos del objeto.

```
"[.RegistroInfantil" <- function(x, i, j, drop) {
  if (i=="id") {
     return(x$id)
  }
  if (i=="datos") {
     return(x$datos)
  }
}
```

De forma similar, se puede definir un método 'set' para sobrecargar el operador '`[`' y acceder a los atributos de la clase por su nombre. El identificador del método se obtiene concatenando el operador '`[`'seguido del operador '`<-`', un punto y el nombre de la clase. En este ejemplo, el identificador es "`[<-.RegistroInfantil`" y la declaración de los argumentos es `function(x, i, j, value)`, que incluye el valor que se asigna al objeto.

```
"[<-.RegistroInfantil" <- function(x, i, j, value) {
  if (i=="id") {
    x$id <- value
  }
  if (i=="datos") {
    x$datos <- value
  }
  return(x)
}
```

El uso de métodos 'get' y 'set' facilita la manipulación de los datos de un objeto.

```
> id <- getId(obj)
> id
[1] 1
> datos <- getDatos(obj)
> datos
  mes estatura  peso perimetro.craneal
1   0    50.10  3.60              35.4
2   3    62.00  6.40              41.5
3   6    67.85  8.15              44.4
4  12    76.15 10.45              47.2
5  18    82.40 11.80              48.4
6  24    87.65 12.75              49.3
> setId(obj) <- 2
> obj["id"]
[1] 2
> datos$peso <- c(3.65, 6.45, 8.20, 10.50, 11.85, 12.80)
> setDatos(obj) <- datos
> obj["id"] <- id
> getId(obj)
[1] 1
> datos$peso <- c(3.65, 6.45, 8.25, 10.50, 11.80, 13.15)
> obj["datos"] <- datos
> print(obj)
id: 1
  mes estatura  peso perimetro.craneal
1   0    50.10  3.65              35.4
2   3    62.00  6.45              41.5
3   6    67.85  8.25              44.4
4  12    76.15 10.50              47.2
5  18    82.40 11.80              48.4
6  24    87.65 13.15              49.3
```

La función `methods(class="class-name")` muestra los métodos definidos para una clase S3.

```
> methods(class="RegistroInfantil")
[1] [.RegistroInfantil          [<-.RegistroInfantil
[3] display.RegistroInfantil    getDatos.RegistroInfantil
[5] getId.RegistroInfantil      plot.RegistroInfantil
[7] print.RegistroInfantil      setDatos<-.RegistroInfantil
[9] setId<-.RegistroInfantil    show.RegistroInfantil
```

### 3.1.3. Funciones de uso común

A continuación se describen las funciones de uso común de las clases S3.

| Función | Descripción |
|---|---|
| `class(x)` | Modifica el atributo class del objeto x |
| `unclass(x)` | Elimina el atributo class del objeto x |
| `methods()` | `methods(class="class-name")` muestra los métodos de una clase S3 |
| `is(object)` | Devuelve la clase del objeto |
| `str(object)` | Devuelve los atributos de un objeto |
| `UseMethod()` | Define un método genérico para la clase |

## 3.2. Clases S4

El sistema de clases S4 ofrece las características básicas de los lenguajes orientados a objetos [6, 8-11]. Dispone de mecanismos de declaración formal de clases, de validación de objetos y de extensión de clases. Las clases S4 se desarrollan de forma similar a como se hace en los lenguajes orientados a objetos. Para definir una nueva clase se utiliza la función `setClass()` y `new()` para instanciar un objeto. La función `setMethod()` define los métodos de la clase. Para saber la clase a la que pertenece un objeto se utiliza la función `class()`.

Los atributos de las clases S4 se denominan 'slots'. Un 'slot' tiene un identificador y un tipo de dato. El contenido de un 'slot' debe coincidir con el tipo de dato con el que ha sido declarado. Para acceder al valor almacenado en cada 'slot' de un objeto se utiliza el carácter `@`.

### 3.2.1. Declaración de clases

La función `setClass(id, representation, validity, contains)` declara una clase. El argumento `id` indica el nombre de la clase, `representation` define la lista de atributos de la clase y sus tipos de datos, `validity` declara la función de validación de los objetos y `contains` se utiliza para declarar una subclase.

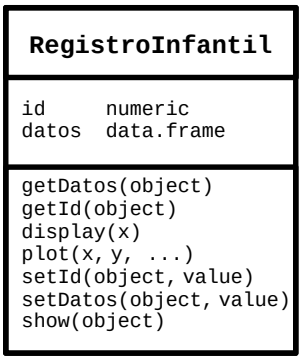
```
setClass("RegistroInfantil",
         representation(id="numeric", datos="data.frame"))
```

La clase `RegistroInfantil` se compone de dos atributos: `id` es el identificador del objeto y `datos` es un `data.frame` que almacena la edad, la estatura, el peso y el perímetro craneal para las seis medidas tomadas desde el nacimiento del niño, hasta que alcanza los 24 meses de edad.

Para definir el objeto de la clase `RegistroInfantil` se utilizan los siguientes datos:

```
> meses <- c(0, 3, 6, 12, 18, 24)
> peso <- c(3.60, 6.40, 8.15, 10.45, 11.80, 12.75)
> estatura <- c(50.10, 62.00, 67.85, 76.15, 82.40, 87.65)
> perimetro.craneal <- c(35.40, 41.50, 44.40, 47.20, 48.40, 49.30)
```

El diagrama de clases muestra los atributos y los métodos de la clase.




Para crear un objeto de la clase se ejecuta la función `new()` con el nombre de la clase, el identificador del registro y el `data.frame`.

```
> obj <- new("RegistroInfantil", id=1,
+             datos=data.frame(meses, estatura, peso, perimetro.craneal))
```

El objeto pertenece a la clase `RegistroInfantil`.

```
> class(obj)
[1] "RegistroInfantil"
attr(,"package")
[1] ".GlobalEnv"
```

El estado del objeto y su clase.

```
>obj
An object of class "RegistroInfantil"
Slot "id":
[1] 1

Slot "datos":
  meses estatura  peso perimetro.craneal
1     0    50.10  3.60              35.4
2     3    62.00  6.40              41.5
3     6    67.85  8.15              44.4
4    12    76.15 10.45              47.2
5    18    82.40 11.80              48.4
6    24    87.65 12.75              49.3
```

La función `str()` también muestra el contenido de un objeto, la clase a la que pertenece, sus atributos y el paquete al que pertenece la clase.

```
> str(obj)
Formal class 'RegistroInfantil' [package ".GlobalEnv"] with 2 slots
  ..@ id   : num 1
  ..@ datos:'data.frame':       6 obs. of  4 variables:
  .. ..$ meses           : num [1:6] 0 3 6 12 18 24
  .. ..$ estatura        : num [1:6] 50.1 62 67.8 76.2 82.4 ...
  .. ..$ peso            : num [1:6] 3.6 6.4 8.15 10.45 11.8 ...
  .. ..$ perimetro.craneal: num [1:6] 35.4 41.5 44.4 47.2 48.4 49.3
```

Las funciones `getClasss()` y `showClass()` devuelven los atributos de la clase y el paquete al que pertenece la clase.

```
> getClass("RegistroInfantil")
Class "RegistroInfantil" [in ".GlobalEnv"]

Slots:

Name:          id      datos
Class:    numeric data.frame
```

La función `getSlots()` devuelve el detalle de atributos de una clase.

```
> getSlots("RegistroInfantil")
          id        datos
   "numeric" "data.frame"
```

Una vez declarada la clase, basta con utilizar la función `new()` para instanciar un objeto. Aunque no es necesario, se recomienda declarar un método constructor para los objetos de la clase. El identificador del método constructor debe coincidir con el identificador de la clase. El método constructor `RegistroInfantil` recibe dos argumentos: `id` es el identificador numérico del objeto y `datos` el `data.frame`.

```
setClass("RegistroInfantil",
         representation(id="numeric", datos="data.frame"))

RegistroInfantil <- function(id, datos) new("RegistroInfantil", id=id,
                                            datos=datos)
```

Para crear un objeto basta con ejecutar el método constructor de la clase.

```
> obj <- RegistroInfantil(id=1, datos=data.frame(meses, estatura,
+                         peso, perimetro.craneal))
```

Además de declarar el método constructor de la clase, se debe declarar una función de validación de datos para que se ejecute cada vez que se instancie un objeto. Para validar una clase se puede utilizar el argumento `validity` de la función `setClass()` o el método `SetValidity()`. Entre estas dos opciones, se recomienda utilizar la función `SetValidity()` porque la declaración de la clase es más sencilla. A continuación se muestran ambos tipos de validación.

Si la validación de la clase utiliza el atributo `validity`, se declara la función de validación `validate.RegistroInfantil` y después se hace referencia a ella en la función `SetClass()`.

```
validate.RegistroInfantil <- function(object) {
  if(!all(sapply(object@datos, is.numeric))) {
    return("Solo se admiten datos numéricos")
  } else return(TRUE)
}

setClass("RegistroInfantil",
         representation(id="numeric", datos="data.frame"),
         validity=validate.registro.infantil)
```

Si la validación de la clase utiliza la función `SetValidity()`, se declara la clase y después el método `SetValidity()` con el nombre de la clase y el código de la función de validación.

```
setClass("RegistroInfantil",
         representation(id="numeric", datos="data.frame"))

setValidity("RegistroInfantil",  function(object) {
  if(!all(sapply(object@datos, is.numeric))) {
    return("Solo se admiten datos numéricos")
  } else return(TRUE)
})
```

El método constructor se declara después de la clase y la función de validación.

```
RegistroInfantil <- function(id, datos) new("RegistroInfantil", id=id,
                                            datos=datos)
```

Durante la instanciación se ejecuta la función de validación y el código de manejo de excepciones de la clase.

```
> obj <- RegistroInfantil(id=1, datos=data.frame(meses, estatura,
+                          peso, c("a", "b", "c", "d", "e", "f")))
Error en validObject(.Object) :
  invalid class “RegistroInfantil” object: Solo se admiten datos numéricos
```

### 3.2.2. Declaración de métodos

Si se ejecuta el método estándar `show` con el objeto `obj`, R muestra la clase a la que pertenece, sus atributos y los valores almacenados en el objeto.

```
> show(obj)
An object of class "RegistroInfantil"
Slot "id":
[1] 1

Slot "datos":
  meses estatura  peso perimetro.craneal
1     0    50.10  3.60              35.4
2     3    62.00  6.40              41.5
3     6    67.85  8.15              44.4
4    12    76.15 10.45              47.2
5    18    82.40 11.80              48.4
6    24    87.65 12.75              49.3
```

Para modificar el comportamiento estándar de un método de R para los objetos de una clase S4, se utiliza la función `setMethod(f, signature, definition)`. El argumento `f` indica el nombre del método, `signature` el nombre de la clase y

`definition` representa el código del método. Por ejemplo, para definir un nuevo método `show` para la clase `RegistroInfantil` se utiliza la declaración `function(object)`, que coincide con el método estándar `show`.

```
setMethod(f="show", signature="RegistroInfantil",
          definition=function(object) {
  cat("id:", object@id, "\n")
  print(object@datos)
})
```

Este método sobreescribe el método `show` de R y la salida es:

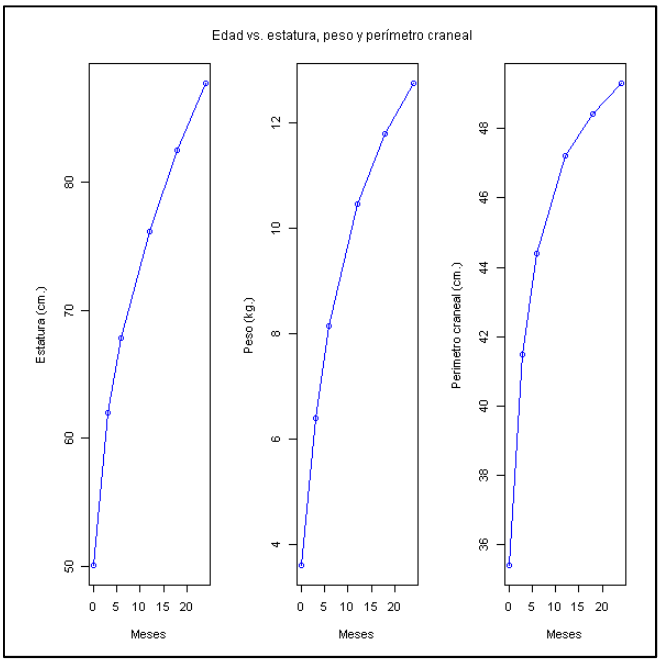
```
> show(obj)
id: 1
  meses estatura  peso perimetro.craneal
1     0    50.10  3.60              35.4
2     3    62.00  6.40              41.5
3     6    67.85  8.15              44.4
4    12    76.15 10.45              47.2
5    18    82.40 11.80              48.4
6    24    87.65 12.75              49.3
```

Para definir un método `plot` para la clase `RegistroInfantil`, se utiliza la función `setMethod()`. Este método muestra tres gráficos para ver la evolución de la estatura, el peso y el diámetro craneal del niño durante sus dos primeros años de vida.

```
setMethod(f="plot", signature="RegistroInfantil",
          definition=function(x, y, ...) {
  par(mfrow=c(1,3), oma=c(0, 2, 0, 0))

  plot(x=x@datos$mes, y=x@datos$estatura, type="o", col="blue",
       xlab="Meses", ylab="Estatura (cm.)")
  plot(x=x@datos$mes, y=x@datos$peso, type="o", col="blue",
       xlab="Meses", ylab="Peso (kg.)")
  plot(x=x@datos$mes, y=x@datos$perimetro.craneal, type="o", col="blue",
       xlab="Meses", ylab="Perimetro craneal (cm.)")

  mtext("Edad vs. estatura, peso y perímetro craneal", side=3, outer=TRUE,
        line=-2.5, cex=0.75)
})
```

El método `plot` construye tres gráficos que muestran la evolución de la estatura, el peso y el diámetro craneal con respecto a la edad del niño.

```
> plot(obj)
```




Para definir un método nuevo se utiliza la función `setGeneric(name, def)`. Primero se declara el nombre del método y después se define el código del método utilizando las funciones `setMethod()` o `setReplaceMethod()`, dependiendo de si el método es de lectura o escritura. Si el método es de lectura y extrae información del objeto se utiliza la función `setMethod(f, signature, definition)`. Si el método es de escritura y modifica el estado del objeto se utiliza la función `setReplaceMethod(f, signature, definition)`. El argumento `definition` de las funciones `setMethod()` y `setReplaceMethod()` representa el código del método. Si el código del método está compuesto de dos o tres expresiones, se puede definir dentro de los métodos `setMethod()` y `setReplaceMethod()` utilizando el argumento `function(object)` y declarando el código dentro de las llaves de la función. Si el código de método es extenso, es mejor definir una nueva función y hacer referencia a ella con el argumento `definition`. El uso de métodos genéricos permite definir métodos 'get' y 'set' para los atributos de la clase.

El método `getId` extrae el valor del atributo `id`.

```
setGeneric("getId", function(object) {
  standardGeneric("getId")
})

setMethod(f="getId", signature="RegistroInfantil",
          definition=function(object) { return(object@id) } )
```

El método `getDatos` extrae el valor del atributo `datos`.

```
setGeneric("getDatos", function(object) {
  standardGeneric("getDatos")
})

setMethod(f="getDatos", signature="RegistroInfantil",
          definition=function(object) { return(object@datos) } )
```

Para definir un método 'set' se utilizan las funciones `setGeneric()` y `setReplaceMethod()`. El identificador del método se concatena con el operador de asignación. Por ejemplo, el método `setId` se declara "`setId<-`". La declaración de los argumentos `function(object, value)` incluye el valor que se asigna al objeto

El método "`setId<-`" modifica el valor del atributo `id`.

```
setGeneric("setId<-", function(object, value) {
  standardGeneric("setId<-")
})

setReplaceMethod(f="setId", signature="RegistroInfantil",
                 definition=function(object, value) {
  object@id <- value
  validObject(object)
  return(object)
})
```

El método "`setDatos<-`" modifica el valor del atributo `datos`.

```
setGeneric("setDatos<-", function(object, value) {
  standardGeneric("setDatos<-")
})

setReplaceMethod(f="setDatos", signature="RegistroInfantil",
                 definition=function(object, value) {
  object@datos <- value
  validObject(object)
  return(object)
})
```

Si se desea, se puede definir un método 'get' para sobrecargar el operador '`[`' y acceder a los atributos de la clase por su nombre. La declaración de los argumentos es `function(x, i, j, drop)` para acceder a los elementos del objeto.

```
setMethod(f="[", signature="RegistroInfantil",
          definition=function(x, i, j, drop) {
  if (i=="id") {
    return(x@id)
  }
  if (i=="datos") {
    return(x@datos)
  }
})
```

De forma similar, se puede definir un método 'set' para sobrecargar el operador '`[`'y acceder a los atributos de la clase por su nombre. La declaración de los argumentos es `function(x, i, j, value)`, que incluye el valor que se asigna al objeto.

```
setReplaceMethod(f="[", signature="RegistroInfantil",
                 definition=function(x, i, j, value) {
  if (i=="id") {
    x@id <- value
  }
  if (i=="datos") {
    x@datos <- value
  }
  validObject(x)
  return(x)
})
```

El uso de métodos ‘get’ y ‘set’ facilita la manipulación de los datos de un objeto.

```
> obj@id
[1] 1
> obj@datos
  meses estatura  peso perimetro.craneal
1     0    50.10  3.60              35.4
2     3    62.00  6.40              41.5
3     6    67.85  8.15              44.4
4    12    76.15 10.45              47.2
5    18    82.40 11.80              48.4
6    24    87.65 12.75              49.3
> id <- getId(obj)
> datos <- getDatos(obj)
> setId(obj) <- 2
> obj["id"]
[1] 2
> datos$peso <- c(3.65, 6.45, 8.20, 10.50, 11.85, 12.80)
> setDatos(obj) <- datos
> obj["id"] <- id
> show(obj)
id: 1
  meses estatura  peso perimetro.craneal
1     0    50.10  3.65              35.4
2     3    62.00  6.45              41.5
3     6    67.85  8.20              44.4
4    12    76.15 10.50              47.2
5    18    82.40 11.85              48.4
6    24    87.65 12.80              49.3
> datos$peso <- c(3.60, 6.40, 8.15, 10.45, 11.80, 12.75)
> obj["datos"] <- datos
> getDatos(obj)
  meses estatura  peso perimetro.craneal
1     0    50.10  3.60              35.4
2     3    62.00  6.40              41.5
3     6    67.85  8.15              44.4
4    12    76.15 10.45              47.2
5    18    82.40 11.80              48.4
6    24    87.65 12.75              49.3
```

La función `showMethods(class="class-name")` muestra los métodos definidos para una clase S4.

```
> showMethods(class="RegistroInfantil")
Function: [ (package base)
x="RegistroInfantil"

Function: [<- (package base)
x="RegistroInfantil"

Function: getDatos (package .GlobalEnv)
object="RegistroInfantil"
```

```
Function: getId (package .GlobalEnv)
object="RegistroInfantil"

Function: initialize (package methods)
.Object="RegistroInfantil"
    (inherited from: .Object="ANY")

Function: plot (package graphics)
x="RegistroInfantil"

Function: setDatos<- (package .GlobalEnv)
object="RegistroInfantil"

Function: setId<- (package .GlobalEnv)
object="RegistroInfantil"

Function: show (package methods)
object="RegistroInfantil"
```

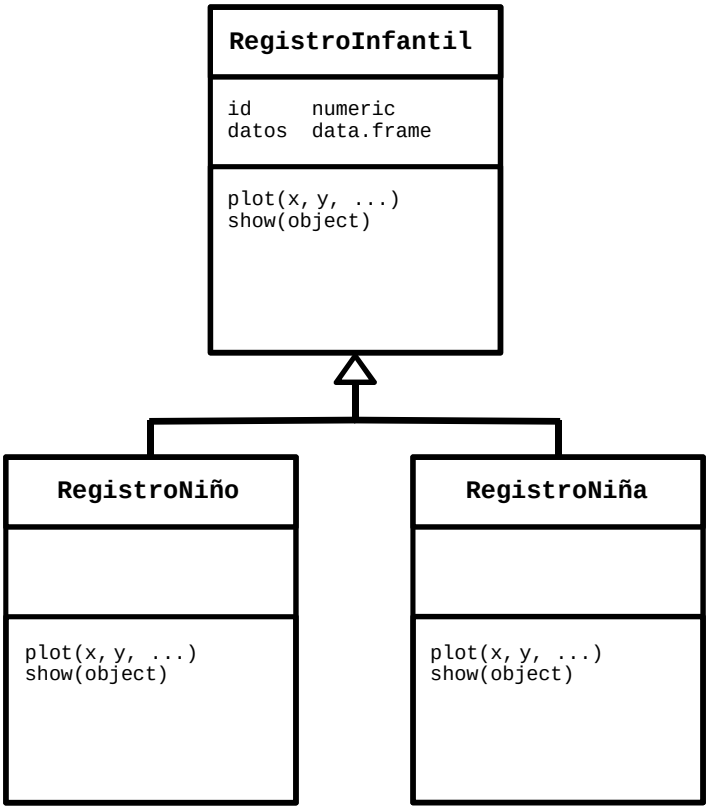
La función `getMethod(f=method, signature=class)` muestra el código de un método.

```
> getMethod(f="show", signature="RegistroInfantil")
Method Definition:

function (object)
{
    cat("id:", object@id, "\n")
    print(object@datos)
}

Signatures:
        object
target  "RegistroInfantil"
defined "RegistroInfantil"
```

### 3.2.3. Extensión de clases

La herencia es la capacidad que tienen los lenguajes orientados a objetos para extender clases. Esto da lugar a una nueva clase que hereda el comportamiento y los atributos de la clase que ha sido extendida. La clase original se denomina clase base o superclase y la nueva clase se denomina clase derivada o subclase. Una subclase hereda los atributos y los métodos de la superclase a la que extiende. Una subclase es una especialización de la superclase y normalmente añade nuevos atributos y métodos que le dan un comportamiento diferente al de la superclase.

Las clases S4 permiten extender otras clases y sobreescribir métodos. La superclase define los atributos y los métodos comunes a todas las subclases. Para mostrar las características de la herencia en las clases S4 se utiliza como base la clase `RegistroInfantil` y se definen las subclases, `RegistroNiño` y `RegistroNiña`, que modifican el comportamiento de los métodos de la superclase.




La declaración de la clase `RegistroInfantil`.

```
setClass("RegistroInfantil",
         representation(id="numeric", datos="data.frame"))

setValidity("RegistroInfantil",  function(object) {
  if(!all(sapply(object@datos, is.numeric))) {
    return("Solo se admiten datos numéricos")
  } else return(TRUE)
})

RegistroInfantil <- function(id, datos) new("RegistroInfantil", id=id,
                                            datos=datos)
```

Los métodos `show` y `plot` de la clase `RegistroInfantil`.

```
setMethod(f="show", signature="RegistroInfantil",
          definition=function(object) {
  cat("id:", object@id, "\n")
  print(object@datos)
})

setMethod(f="plot", signature="RegistroInfantil",
          definition=function(x, y, ...) {
  par(mfrow=c(1,3), oma=c(0, 2, 0, 0))

  plot(x=x@datos$mes, y=x@datos$estatura, type="o", col="blue",
       xlab="Meses", ylab="Estatura (cm.)")
  plot(x=x@datos$mes, y=x@datos$peso, type="o", col="blue",
       xlab="Meses", ylab="Peso (kg.)")
  plot(x=x@datos$mes, y=x@datos$perimetro.craneal, type="o", col="blue",
       xlab="Meses", ylab="Perimetro craneal (cm.)")

  mtext("Edad vs. estatura, peso y perímetro craneal", side=3, outer=TRUE,
        line=-2.5, cex=0.75)
})
```

Para indicar que las clases `RegistroNiño` y `RegistroNiña` se derivan de la clase `RegistroInfantil` se utiliza el argumento `contains` y se hacer referencia al identificador de la superclase. Las subclases de este ejemplo no definen nuevos atributos, solo sobreescriben los métodos `show` y `plot` de la clase superclase. Los métodos `show` y `plot` de las subclases indican si los datos corresponden a un niño o una niña. Los gráficos de los niños utilizan el color azul marino y los gráficos de las niñas el color rojo.

La clase `RegistroNiño` y sus métodos `show` y `plot`.

```
setClass("RegistroNiño", contains="RegistroInfantil")

RegistroNiño <- function(id, datos) new("RegistroNiño", id=id, datos=datos)

setMethod(f="show", signature="RegistroNiño", definition=function(object) {
  cat("id:", object@id, "\t", "Niño", "\n")
  print(object@datos)
})

setMethod(f="plot", signature="RegistroNiño",
          definition=function(x, y, ...) {
  par(mfrow=c(1,3), oma=c(0, 2, 0, 0))

  plot(x=x@datos$mes, y=x@datos$estatura, type="o", col="navy",
       xlab="Meses", ylab="Estatura (cm.)")
  plot(x=x@datos$mes, y=x@datos$peso, type="o", col="navy",
       xlab="Meses", ylab="Peso (kg.)")
  plot(x=x@datos$mes, y=x@datos$perimetro.craneal, type="o", col="navy",
       xlab="Meses", ylab="Perimetro craneal (cm.)")

  mtext("Edad vs. estatura, peso y perímetro craneal de un niño", side=3,
        outer=TRUE, line=-2.5, cex=0.75)
})
```

La clase `RegistroNiña` y sus métodos `show` y `plot`.

```
setClass("RegistroNiña", contains="RegistroInfantil")

RegistroNiña <- function(id, datos) new("RegistroNiña", id=id, datos=datos)

setMethod(f="show", signature="RegistroNiña", definition=function(object) {
  cat("id:", object@id, "\t", "Niña", "\n")
  print(object@datos)
})

setMethod(f="plot", signature="RegistroNiña",
          definition=function(x, y, ...) {
  par(mfrow=c(1,3), oma=c(0, 2, 0, 0))

  plot(x=x@datos$mes, y=x@datos$estatura, type="o", col="red",
       xlab="Meses", ylab="Estatura (cm.)")
  plot(x=x@datos$mes, y=x@datos$peso, type="o", col="red",
       xlab="Meses", ylab="Peso (kg.)")
  plot(x=x@datos$mes, y=x@datos$perimetro.craneal, type="o", col="red",
       xlab="Meses", ylab="Perimetro craneal (cm.)")

  mtext("Edad vs. estatura, peso y perímetro craneal de una niña", side=3,
        outer=TRUE, line=-2.5, cex=0.75)
})
```

Para definir los objetos de las clases `RegistroInfantil`, `RegistroNiño` y `RegistroNiña` se utilizan los siguientes datos:

```
> meses <- c(0, 3, 6, 12, 18, 24)
> peso.niño <- c(3.60, 6.40, 8.15, 10.45, 11.80, 12.75)
> estatura.niño <- c(50.10, 62.00, 67.85, 76.15, 82.40, 87.65)
> perimetro.craneal.niño <- c(35.40, 41.50, 44.40, 47.20, 48.40, 49.30)
> peso.niña <- c(3.10, 5.89, 7.80, 9.60, 10.70, 11.30)
> estatura.niña <- c(47.20, 59.50, 67.86, 76.11, 82.41, 87.66)
> perimetro.craneal.niña <- c(34.50, 40.20, 43.10, 46.50, 47.20, 48.50)
```

Creación de objetos de las clases con los datos correspondientes a un niño y una niña.

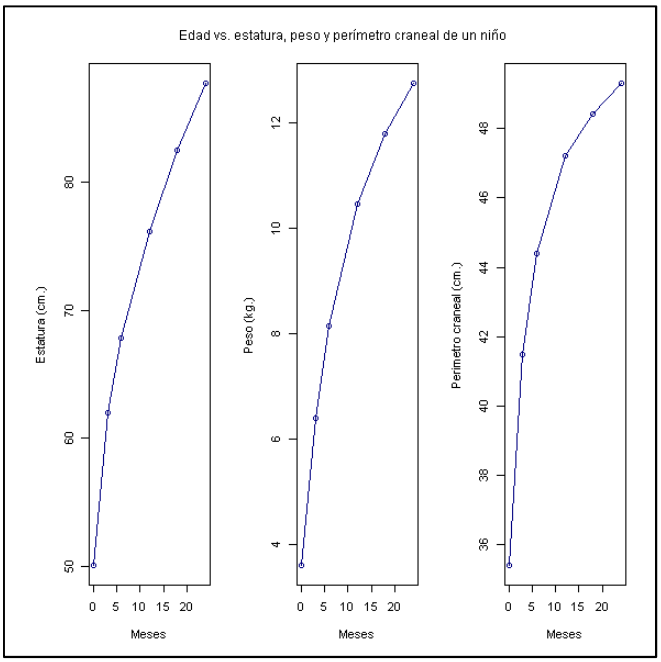
```
> obj <- RegistroInfantil(id=1, datos=data.frame(meses, estatura.niño,
+                          peso.niño, perimetro.craneal.niño))
> niño <- RegistroNiño(id=2, datos=data.frame(meses, estatura.niño,
+                       peso.niño, perimetro.craneal.niño))
> niña <- RegistroNiña(id=3, datos=data.frame(meses, estatura.niña,
+                       peso.niña, perimetro.craneal.niña))
```

Cuando se ejecutan los métodos `show` o `plot`, R comprueba la clase a la que pertenece el objeto y ejecuta el método de la clase base o de la clase derivada, según corresponda.

```
> show(obj)
id: 1
  meses estatura.niño peso.niño perimetro.craneal.niño
1     0         50.10      3.60                   35.4
2     3         62.00      6.40                   41.5
3     6         67.85      8.15                   44.4
4    12         76.15     10.45                   47.2
5    18         82.40     11.80                   48.4
6    24         87.65     12.75                   49.3
> show(niño)
id: 2    Niño
  meses estatura.niño peso.niño perimetro.craneal.niño
1     0         50.10      3.60                   35.4
2     3         62.00      6.40                   41.5
3     6         67.85      8.15                   44.4
4    12         76.15     10.45                   47.2
5    18         82.40     11.80                   48.4
6    24         87.65     12.75                   49.3
> show(niña)
id: 3    Niña
  meses estatura.niña peso.niña perimetro.craneal.niña
1     0         47.20      3.10                   34.5
2     3         59.50      5.89                   40.2
3     6         67.86      7.80                   43.1
4    12         76.11      9.60                   46.5
5    18         82.41     10.70                   47.2
6    24         87.66     11.30                   48.5
```

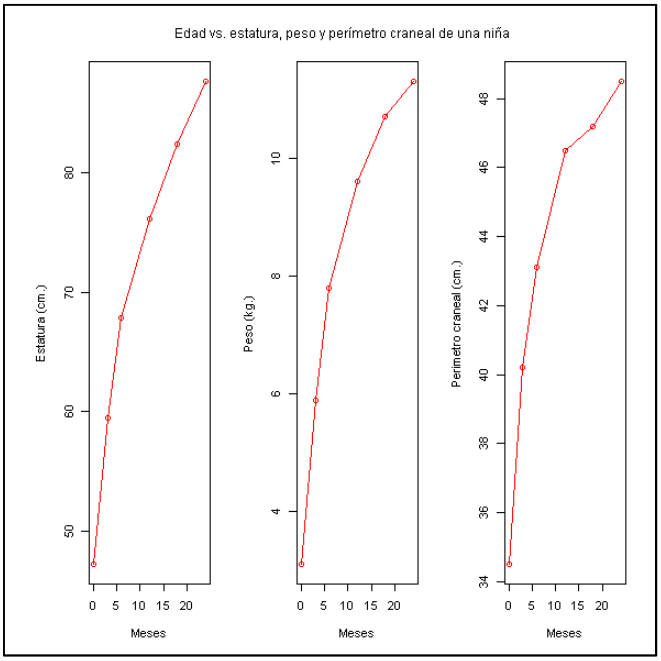
El método `plot` de la clase `RegistroNiño` muestra los gráficos en azul marino.

```
> plot(niño)
```




El método `plot` de la clase `RegistroNiña` muestra los gráficos en rojo.

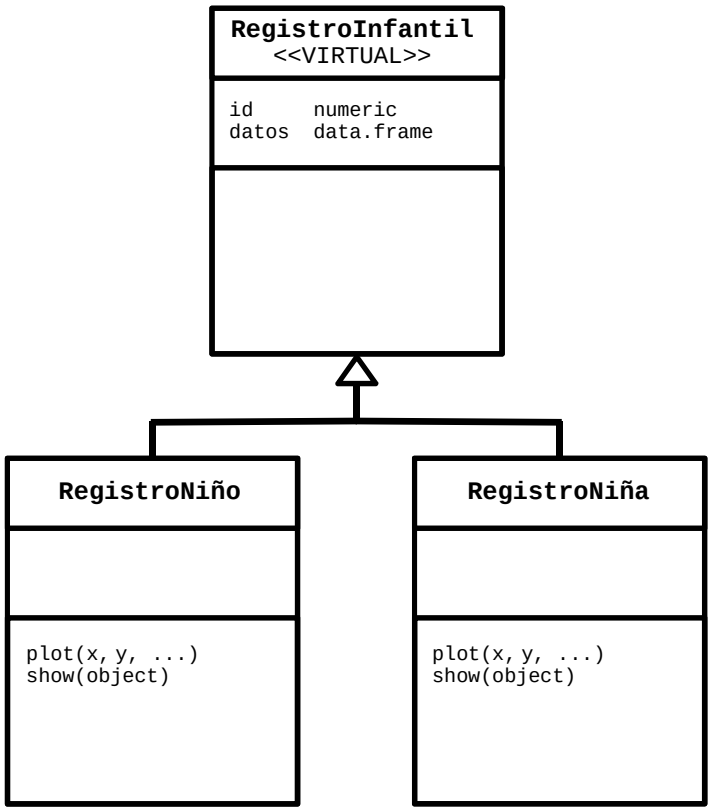
```
> plot(niña)
```



### 3.2.4. Clases virtuales

El sistema de clases S4 permite definir clases abstractas, es decir, clases que no pueden ser instanciadas pero que se pueden utilizar para definir otras clases. Las clases abstractas de S4 se denominan clases virtuales y se definen con la etiqueta `VIRTUAL`.

A continuación se desarrolla el mismo ejemplo de la superclase `RegistroInfantil` y las subclases `RegistroNiño` y `RegistroNiña` utilizando clases virtuales. La superclase `RegistroInfantil` se define como `VIRTUAL` y no tiene métodos propios. Las subclases `RegistroNiño` y `RegistroNiña` definen sus métodos `show` y `plot`. En este diseño solo permite crear objetos de las clases `RegistroNiño` y `RegistroNiña`. Dado que los métodos `show` y `plot` de las subclases tienen características particulares, no se definen métodos en la superclase para que sean las subclases las responsables de definir el comportamiento de los objetos.




La declaración de la clase virtual `RegistroInfantil`.

```
setClass("RegistroInfantil",
         representation(id="numeric", datos="data.frame", "VIRTUAL"))

setValidity("RegistroInfantil",  function(object) {
  if(!all(sapply(object@datos, is.numeric))) {
    return("Solo se admiten datos numéricos")
  } else return(TRUE)
})
```

```
RegistroInfantil <- function(id, datos) new("RegistroInfantil", id=id,
                                            datos=datos)
```

La clase `RegistroNiño` y sus métodos `show` y `plot`.

```
setClass("RegistroNiño", contains="RegistroInfantil")

RegistroNiño <- function(id, datos) new("RegistroNiño", id=id, datos=datos)

setMethod(f="show", signature="RegistroNiño", definition=function(object) {
  cat("id:", object@id, "\t", "Niño", "\n")
  print(object@datos)
})

setMethod(f="plot", signature="RegistroNiño",
          definition=function(x, y, ...) {
  par(mfrow=c(1,3), oma=c(0, 2, 0, 0))

  plot(x=x@datos$mes, y=x@datos$estatura, type="o", col="navy",
       xlab="Meses", ylab="Estatura (cm.)")
  plot(x=x@datos$mes, y=x@datos$peso, type="o", col="navy",
       xlab="Meses", ylab="Peso (kg.)")
  plot(x=x@datos$mes, y=x@datos$perimetro.craneal, type="o", col="navy",
       xlab="Meses", ylab="Perimetro craneal (cm.)")

  mtext("Edad vs. estatura, peso y perímetro craneal de un niño", side=3,
        outer=TRUE, line=-2.5, cex=0.75)
})
```

La clase `RegistroNiña` y sus métodos `show` y `plot`.

```
setClass("RegistroNiña", contains="RegistroInfantil")

RegistroNiña <- function(id, datos) new("RegistroNiña", id=id, datos=datos)


setMethod(f="show", signature="RegistroNiña", definition=function(object) {
  cat("id:", object@id, "\t", "Niña", "\n")
  print(object@datos)
})

setMethod(f="plot", signature="RegistroNiña",
          definition=function(x, y, ...) {
  par(mfrow=c(1,3), oma=c(0, 2, 0, 0))

  plot(x=x@datos$mes, y=x@datos$estatura, type="o", col="red",
       xlab="Meses", ylab="Estatura (cm.)")
  plot(x=x@datos$mes, y=x@datos$peso, type="o", col="red",
       xlab="Meses", ylab="Peso (kg.)")
  plot(x=x@datos$mes, y=x@datos$perimetro.craneal, type="o", col="red",
       xlab="Meses", ylab="Perimetro craneal (cm.)")

  mtext("Edad vs. estatura, peso y perímetro craneal de una niña", side=3,
        outer=TRUE, line=-2.5, cex=0.75)
})
```

La clase `RegistroInfantil` no se puede instanciar, de manera que solo se pueden crear objetos de las clases `RegistroNiño` y `RegistroNiña`.

```
> niño <- RegistroNiño(id=1, datos=data.frame(meses, estatura.niño,
+                       peso.niño, perimetro.craneal.niño))
> niña <- RegistroNiña(id=2, datos=data.frame(meses, estatura.niña,
+                       peso.niña, perimetro.craneal.niña))
```

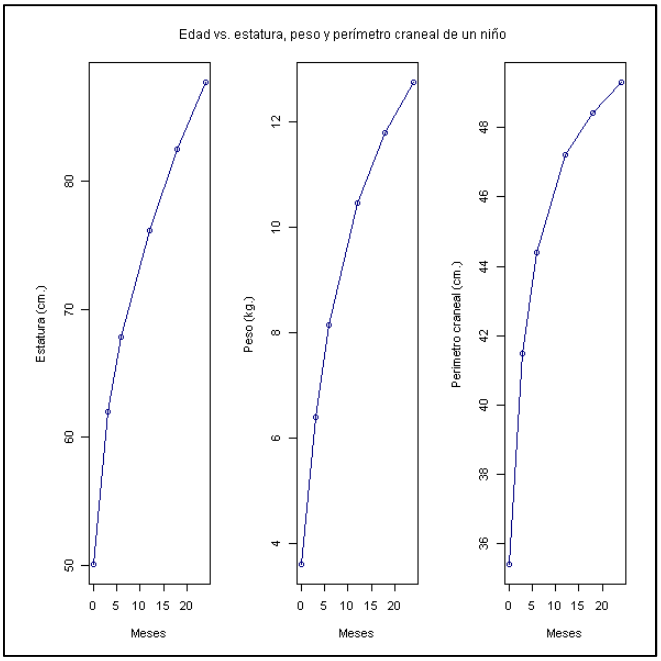
Si se intenta crear un objeto de la clase `RegistroInfantil` se produce un error.

```
> obj <- RegistroInfantil(id=1, datos=data.frame(meses, estatura.niño,
+                         peso.niño, perimetro.craneal.niño))
Error en new("RegistroInfantil", id = id, datos = datos) :
  trying to generate an object from a virtual class ("RegistroInfantil")
```

Cuando se ejecutan los métodos `show` o `plot`, R comprueba la clase a la que pertenece el objeto y ejecuta el método de la clase correspondiente.

```
> show(niño)
id: 1    Niño
  meses estatura.niño peso.niño perimetro.craneal.niño
1     0         50.10      3.60                   35.4
2     3         62.00      6.40                   41.5
3     6         67.85      8.15                   44.4
4    12         76.15     10.45                   47.2
5    18         82.40     11.80                   48.4
6    24         87.65     12.75                   49.3
> show(niña)
id: 2    Niña
  meses estatura.niña peso.niña perimetro.craneal.niña
1     0         47.20      3.10                   34.5
2     3         59.50      5.89                   40.2
3     6         67.86      7.80                   43.1
4    12         76.11      9.60                   46.5
5    18         82.41     10.70                   47.2
6    24         87.66     11.30                   48.5
```

El método `plot` de la clase `RegistroNiño` muestra los gráficos en azul marino y el método `plot` de la clase `RegistroNiña` muestra los gráficos en rojo.



### 3.2.5. Funciones de uso común

A continuación se describen las funciones de uso común de las clases S4.

| Función | Descripción |
|---|---|
| setClass() | Define una clase |
| setMethod() | Define un método de una clase |
| setGeneric() | Define una 'función genérica' |
| new() | Instancia un objeto |
| getClass() | Devuelve la clase a la que pertenece un objeto |
| getMethod() | Muestra el código de un método, dado su identificador y la clase a la que pertenece |
| showMethods() | showMethods(class="class-name") muestra los métodos de una clase S4 |
| getSlots() | Devuelve los atributos de una clase |
| @ | Accede al valor almacenado en un atributo |
| validObject() | Comprueba la validez de un objeto |

# Capítulo 4. Desarrollo de librerías en C para R

R es un lenguaje interpretado, esto afecta directamente al rendimiento de las aplicaciones y puede ser una limitación importante cuando se realizan cálculos con grandes cantidades de datos. Para desarrollar aplicaciones más eficientes, es necesario desarrollar librerías en C para integrarlas en las aplicaciones R.

Antes de desarrollar una librería externa, es necesario analizar las ventajas que aporta y sus inconvenientes. El uso de librerías C mejora el rendimiento de las aplicaciones R, pero su desarrollo requiere conocimientos más avanzados de programación y un esfuerzo mayor. Además, el uso de librerías externas dificulta la depuración y las pruebas de las aplicaciones, así que, ¿merece la pena el esfuerzo? Esta pregunta hay que responderla dependiendo de los requisitos de uso de cada aplicación.

Este apartado describe el uso de la función `.C` de R como 'interface' para ejecutar librerías externas [20, 24]. Muestra el uso de vectores de tipo `integer`, `double`, `character` y `logical` en R y el paso de estos objetos a una librería C. Además compara el rendimiento de una función nativa de R con una función que utiliza una librería C y una función completamente desarrollada en R.

## 4.1. La función .C

El primer argumento de la función `.C` es el identificador de la función externa C que se desea ejecutar y a continuación se indica el resto de argumentos. Para cada argumento de la función se indica el tipo de dato de R utilizando las funciones `as.logical`, `as.integer`, `as.double` y `as.character`, según sea necesario. De esta manera, la función externa C interpreta correctamente los datos que recibe.

En el siguiente ejemplo, la función externa `ccov` calcula la covarianza de los vectores `x`, `y`, ambos de tipo `double`. Esta función recibe cuatro argumentos: `x` es de tipo `double`; `y` es de tipo `double`; `n` es de tipo `integer` y almacena la longitud del vector `x`; `sxy` es de tipo `double` y se utiliza como valor de retorno de la función.

```
.C("ccov", x=as.double(x), y=as.double(y),n=as.integer(length(x)),
           sxy=as.double(1))
```

Es muy importante realizar una validación previa de los argumentos antes de ejecutar la función externa C. Utilizando el ejemplo anterior, la función R que llama a la función externa `ccov` debe verificar si los vectores `x`, `y` tienen valores `NA` antes de realizar la llamada.

La función `.C` devuelve un objeto de tipo `List` con todos sus argumentos. En este ejemplo, `sxy` representa el valor de retorno de la función y está almacenado en el objeto `out$sxy`.

```
out <- .C("ccov", x=as.double(x), y=as.double(y),n=as.integer(length(x)),
                  sxy=as.double(1))
```

## 4.2. El código C

Cuando se utiliza la llamada `.C`, el valor de retorno de las funciones de la librería C se declara `void`.

```
//---------------------------------------------------------------------------
// LibC.c
//---------------------------------------------------------------------------

#include <windows.h>
#define DLL_EXPORT __declspec(dllexport)

DLL_EXPORT void ccov(double *x, double *y, int *n, double *sxy) {
  double xm=0.0, ym=0.0;

  // calculo de la media de los vectores x, y

  for (int i=0; i < *n; i++) {
    xm+=x[i];
    ym+=y[i];
  }

  xm/=*n;
  ym/=*n;

  // calculo de la covarianza sxy

  *sxy=0.0;

  for (int i=0; i < *n; i++)
    *sxy+=(x[i]*y[i]) - (xm*ym);

  *sxy/=(*n-1);
}
```

La función externa solo recibe punteros y la manipulación de estos argumentos en el código C depende de si se trata de punteros a vectores, a cadenas de caracteres o a una variable de tipo `integer`, `double` o `character`. Es importante señalar que, antes de realizar la llamada a una función externa C, R hace una copia de los objetos que se utilizan como argumentos en la función y pasa a C los punteros a estos objetos. Esto significa que la función C no puede modificar el valor de los objetos originales de R porque utiliza un espacio de memoria distinto.

Para compilar la librería en Windows se inicia una sesión de la consola del sistema operativo `MS-DOS` y se ejecuta el comando `R CMD SHLIB` seguido del nombre del fichero C. Por ejemplo, para compilar el fichero `LibC.c` y obtener el fichero `DLL`, se

ejecuta `R CMD SHLIB LibC.c`. El comando `R CMD` es parte del conjunto de aplicaciones de `RTools`, como se indica con más detalle en el apartado 6.3.

## 4.3. El código R

La función R `CovarianzaC` hace una llamada a la función C `ccov` almacenada en la librería `LibC`. Esta función recibe como argumentos los vectores `x`, `y` de tipo `double`, `n` de tipo `integer` para indicar la longitud de los vectores y `sxy` de tipo `double`, que representa el valor de retorno de la función.

La función `CovarianzaC` debe verificar que los argumentos son válidos antes de realizar la llamada a la función C. Esta función declara la variable `out` que almacena la copia de los objetos R que se han pasado a la función externa.

```
CovarianzaC <- function(x, y) {
  n <- length(x)

  if (n <= 1)
    stop("¡El número de elementos del vector x debe ser mayor de 1!")
  else
    if (n != length(y))
      stop("¡Los vectores x, y deben tener el mismo númeo de elementos!")

      if (TRUE %in% is.na(x) || TRUE %in% is.na(y)) {
        stop("!Los vectores x, y no pueden tener valores NA!")
      }

  if (!is.loaded("ccov"))
    dyn.load("LibC.dll")

  out <- .C("ccov", x=as.double(x), y=as.double(y), n=as.integer(length(x)),
                    sxy=as.double(1))

  return(out$sxy)
}
```

Los resultados de la función:

```
> x <- c(70,65,85,60,70,75,90,80,60,70)
> y <- c(175,160,180,155,165,180,185,175,160,170)
> CovarianzaC(x, y)
[1] 93.05556
> cov(x, y)
[1] 93.05556
```

## 4.4. Uso de funciones externas C y eficiencia

Para comparar la mejora del rendimiento de las aplicaciones R con el uso de funciones externas C, se desarrolla una nueva función para calcular la covarianza en R, sin utilizar la función `cov` o la librería externa C.

```
CovarianzaR <- function(x, y) {
  n <- length(x)

  if (n <= 1)
    stop("¡El número de elementos del vector x debe ser mayor de 1!")
  else
    if (n != length(y))
      stop("¡Los vectores x, y deben tener el mismo númeo de elementos!")

      if (TRUE %in% is.na(x) || TRUE %in% is.na(y)) {
        stop("!Los vectores x, y no pueden tener valores NA!")
      }

  xm <- mean(x)
  ym <- mean(y)

  sxy <- 0

  i <- as.integer(1)

  while (i <= n) {
    sxy <- sxy + (x[i]*y[i] - xm*ym)

    i <- i + 1
  }

  sxy <- sxy / (n-1)

  return(sxy)
}
```

Los resultados de la función:

```
> CovarianzaR(x, y)
[1] 93.05556
```

La función `TestCovarianza` calcula el tiempo de ejecución de las tres funciones:

```
TestCovarianza <- function(x, y) {
  sxy <- array(0, dim=c(3))
  elapsed <- array(0, dim=c(3))

  start <- proc.time()[3]
  sxy[1] <- cov(x, y)
  elapsed[1] <- proc.time()[3]-start

  start <- proc.time()[3]
  sxy[2] <- CovarianzaC(x, y)
  elapsed[2] <- proc.time()[3]-start

  start <- proc.time()[3]
  sxy[3] <- CovarianzaR(x, y)
  elapsed[3] <- proc.time()[3]-start

  return(list(elapsed=elapsed, sxy=sxy))
}
```

Para comparar el rendimiento de las funciones se utiliza la función `Test(n, size)`, que ejecuta `n` veces las funciones de cálculo de la covarianza con dos vectores de tamaño `size`. Esta función devuelve un objeto que almacena la variable `size`, la matriz `elapsed`, la matriz `sxy` y el vector `avg`. Las matrices `elapsed` y `sxy` tienen `n` filas y tres columnas, almacenan los tiempos de ejecución cada función y el resultado de la

covarianza, respectivamente. El vector avg almacena el tiempo medio de ejecución de cada función.

```
Test <- function(n, size) {
  sxy <- matrix(0, nrow=n, ncol=3,
                dimnames=list(c(1:n), c("cov", "ccov", "R")))
  elapsed <- matrix(0, nrow=n, ncol=3,
                    dimnames=list(c(1:n), c("cov", "ccov", "R")))

  for (i in 1:n) {
    t <- TestCovarianza(sample(1:500, size, replace=TRUE),
                        sample(1:750, size, replace=TRUE))

    sxy[i, ] <- t$sxy
    elapsed[i, ] <- t$elapsed
  }

  obj <- list(size=size, elapsed=elapsed,
              avg=apply(elapsed, 2, mean), sxy=sxy)

  return(obj)
}
```

Para comparar el rendimiento de las funciones se realizan pruebas con vectores de tamaño 1E3, 1E4, 1E5, 1E6 y 1E7. Los resultados para órdenes de magnitud 1E3 y 1E4 no son significativos. Por el contrario, para vectores con tamaño 1E5 o mayor sí se obtienen resultados concluyentes.

A continuación se muestran ejemplos de los resultados para órdenes de magnitud 1E5, 1E6 y 1E7.

```
> test1e5 <- Test(5, 100000)
> test1e5
$size
[1] 1e+05

$elapsed
   cov ccov    R
1 0.03 0.03 0.70
2 0.02 0.01 0.71
3 0.01 0.02 0.67
4 0.02 0.01 0.67
5 0.02 0.01 0.69

$avg
  cov  ccov     R
0.020 0.016 0.688

$sxy
         cov       ccov          R
1  56.493855  56.493855  56.493855
2  -9.222588  -9.222588  -9.222588
3  56.581318  56.581318  56.581318
4   5.105722   5.105722   5.105722
5 -59.723651 -59.723651 -59.723651
```

```
> test1e6 <- Test(5, 1000000)
> test1e6
$size
[1] 1e+06

$elapsed
   cov ccov    R
1 0.27 0.17 6.66
2 0.17 0.14 6.64
3 0.19 0.14 6.64
4 0.19 0.14 6.65
5 0.19 0.19 6.65

$avg
  cov  ccov     R
0.202 0.156 6.648

$sxy
        cov      ccov         R
1  22.05195  22.05195  22.05195
2 -19.15298 -19.15298 -19.15298
3  10.57720  10.57720  10.57720
4  11.14452  11.14452  11.14452
5 -58.93871 -58.93871 -58.93871


> test1e7 <- Test(5, 10000000)
> test1e7
$size
[1] 1e+07

$elapsed
   cov ccov     R
1 2.07 1.47 66.59
2 1.83 1.45 66.48
3 1.85 1.48 66.78
4 1.86 1.47 66.60
5 1.86 1.45 68.34

$avg
   cov   ccov      R
 1.894  1.464 66.958

$sxy
          cov        ccov           R
1   0.2870573   0.2870573   0.2870573
2   7.5934342   7.5934342   7.5934342
3  -7.6877946  -7.6877946  -7.6877946
4 -14.8428329 -14.8428329 -14.8428329
5   4.7133641   4.7133641   4.7133641
```

La siguiente tabla muestra el tiempo medio de ejecución de 5 repeticiones de esta prueba.

| size | cov | ccov | R |
|---|---|---|---|
| 1E5 | 0,020 | 0,016 | 0,688 |
| 1E5 | 0,014 | 0,010 | 0,682 |
| 1E5 | 0,022 | 0,014 | 0,680 |
| 1E5 | 0,018 | 0,018 | 0,676 |
| 1E5 | 0,022 | 0,020 | 0,680 |
| 1E6 | 0,202 | 0,156 | 6,648 |

| size | cov | ccov | R |
|---|---|---|---|
| 1E6 | 0,192 | 0,126 | 6,634 |
| 1E6 | 0,188 | 0,126 | 6,638 |
| 1E6 | 0,184 | 0,128 | 6,630 |
| 1E6 | 0,190 | 0,134 | 6,644 |
| 1E7 | 1,894 | 1,464 | 66,958 |
| 1E7 | 1,850 | 1,438 | 66,048 |
| 1E7 | 1,874 | 1,446 | 66,750 |
| 1E7 | 1,870 | 1,454 | 66,964 |
| 1E7 | 1,856 | 1,442 | 66,130 |

De los resultados anteriores se puede ver que, para órdenes de magnitud 1E5, 1E6 y 1E7, la función externa `ccov` es más eficiente que la función nativa `cov` de R. A continuación se muestra el tiempo medio de los resultados de la tabla anterior.

| size | cov | ccov | R |
|---|---|---|---|
| 1E5 | 0,0192 | 0,0156 | 0,6812 |
| 1E6 | 0,1912 | 0,1340 | 6,6388 |
| 1E7 | 1,8688 | 1,4488 | 66,5700 |

Si se toma como referencia la función más eficiente y se calcula la proporción de los tiempos de ejecución de las otras funciones con respecto a ésta, se tiene.

| size | cov | ccov | R |
|---|---|---|---|
| 1E5 | 1,2308 | 1,0000 | 43,6667 |
| 1E6 | 1,4269 | 1,0000 | 49,5433 |
| 1E7 | 1,2899 | 1,0000 | 45,9484 |

La conclusión es clara, para órdenes de magnitud 1E5 o mayores, merece la pena desarrollar una función externa C. De estos resultados se ve que la función nativa C es entre un 23% y un 42% más lenta que la función externa C. En cuanto a la función R, es evidente que se deben evitar los procesos iterativos, ya que son entre un 4300% y un 4900% más lentos.

## 4.5. Conversión de tipos de datos entre R y C

La siguiente tabla muestra la conversión de tipos de datos entre R y C.

| Tipo de dato en R (storage mode) | Declaración en la función .C de R | Declaración en la librería externa C |
|---|---|---|
| logical | as.logical(n) | int *n |
| integer | as.integer(n) | int *n |
| double | as.double(n) | double *n |
| character | as.character(n) | char **n |

El siguiente ejemplo muestra una función C que manipula vectores de distintos tipos de datos [15]. El vector `x`, de tipo `double` se multiplica por 2; el vector `y`, de tipo `integer` se eleva al cuadrado; el carácter `c` se convierte a minúsculas, el vector `s`, de tipo `character` se convierte a mayúsculas y el vector `b`, de tipo `logical` se le aplica el operador de negación.

```
//-------------------------------------------------------------------------
// LibC.c
//-------------------------------------------------------------------------

#include <windows.h>
#define DLL_EXPORT __declspec(dllexport)

DLL_EXPORT void ctypes(double *x,
                       int *y,
                       char **c,
                       char **s,
                       int *b,
                       int *n) {

   for (int i=0; i < *n; i++) {
      x[i] = x[i]*2.0;
      y[i] = y[i]*y[i];

      *c[0]=tolower(*c[0]);

      char *str = s[i];

      int j=0;

      while (str[j] != '\0') {
         str[j] = toupper(str[j]);
         j++;
      }

      b[i] = !b[i];
   }
}
```

La función R `SampleOfTypes` hace una llamada a la función C `ctypes` de la librería `LibC`.

```
SampleOfTypes <- function(x, y, c, s, b, n) {
  if (!is.loaded("ctypes"))
    dyn.load("LibC.dll")

  out <- .C("ctypes", x=as.double(x),
                      y=as.integer(y),
                      c=as.character(c),
                      s=as.character(s),
                      b=as.logical(b),
                      n=as.integer(length(x)))

  return(out)
}
```

Los resultados de la función:

```
> x <- c(2.0, 3.0, 2.5, 5.0, 10.0)
> y <- c(2, 4, 6, 8, 10)
> c <- "C"
> s <- c("lunes", "martes", "miercoles", "jueves", "viernes")
> b <- c("TRUE", "FALSE", "TRUE", "FALSE", "FALSE")
> SampleOfTypes(x, y, c, s, b, length(x))
$x
[1]  4  6  5 10 20

$y
[1]   4  16  36  64 100

$c
[1] "c"

$s
[1] "LUNES"     "MARTES"    "MIERCOLES" "JUEVES"    "VIERNES"

$b
[1] FALSE  TRUE FALSE  TRUE  TRUE

$n
[1] 5
```

# Capítulo 5. Recomendaciones de estilo para R

A diferencia de muchos lenguajes de programación, R no dispone de una guía de estilo oficial, ni siquiera un documento de recomendaciones de programación que sea ampliamente aceptado [1]. No hay un estándar de codificación reconocido por el equipo de desarrollo de R ni por la comunidad de usuarios. No obstante, existen varias referencias que buscan definir un conjunto de recomendaciones de programación en R, entre las que cabe destacar: "Google's R Style Guide"[12], "R Coding Conventions"[13], "Bioconductor Coding Style"[14], "R Style Guide"[15] y "R Code Conventions"[16].

"Google's R Style Guide" es resultado de la colaboración de la comunidad de usuarios de R de Google; "R Coding Conventions" es un trabajo de Henrik Bengtsson, del Departamento de Estadística de la Universidad de California en Berkeley; "Bioconductor Coding Style" es la propuesta del equipo de desarrollo de Bioconductor; "R Style Guide" es un trabajo de Hadley Wickman, de la Universidad Rice, en Houston Texas; "R Code Conventions" es la propuesta del equipo de desarrollo de CellNOpt. Todas estas guías tienen como objetivo definir recomendaciones de estilo y programación para facilitar el desarrollo y mantenimiento de programas R. Desafortunadamente, existen diferencias notables entre ellas y ni siquiera hay un criterio único aplicable para los identificadores, las clases o las funciones de un programa R.

A continuación se proponen recomendaciones de codificación de R basadas en las guías citadas anteriormente [3, 4, 14, 21, 30].

## 5.1. Identificadores y nombres

El estilo de escritura 'CamelCase' se utiliza para unir palabras sin espacios para formar palabras compuestas o frases. Se basa en el uso de letras mayúsculas y minúsculas para diferenciar las palabras que forman parte de la palabra compuesta. La primera letra de cada palabra se escribe con mayúsculas para hacer más legible la expresión y facilitar su lectura. Existen dos estilos 'CamelCase', el 'UpperCamelCase' comienza con una letra

[12] Google´s R Style Guide
http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html
[13] R Coding Conventions (Henrik Bengtsson, Universidad de California en Berkeley)
https://docs.google.com/document/d/1esDVxyWvH8AsX-VJa-8oqWaHLs4stGlIbk8kLc5VlII/edit?pli=1
[14] Bioconductor Coding Style
http://www.bioconductor.org/developers/coding-style/
[15] R Style Guide (Hadley Wickman, Universidad Rice, Houston, Texas)
http://stat405.had.co.nz/r-style.html
[16] R Code Conventions (CellNOpt)
http://www.cellnopt.org/doc/cnodocs/R_code_layout.html

mayúscula y el ‘lowerCamelCase’ comienza con una letra minúscula. Estos estilos se aplican a nombres de variables y de funciones.

### 5.1.1. Archivos

Los nombres de archivos deben tener la extensión `.R` y su identificador debe ser significativo. El nombre del archivo se debe escribir aplicando el estilo ‘UpperCamelCase’ o utilizando letras minúsculas y el carácter ‘-‘ para separar palabras.

```
# correcto
carga-datos.R
CargaDatos.R

# incorrecto
cdatos.R
```

### 5.1.2. Variables

Los nombres de las variables se deben escribir aplicando el estilo ‘lowerCamelCase’.

```
# correcto
totalMuestras

# incorrecto
total_muestras
```

Para variables de tipo lógico se recomienda utilizar un prefijo para facilitar la interpretación del código. Si los nombres de las variables se escriben en español, se debe utilizar el prefijo ‘es’, si los nombres están en inglés, se debe utilizar el prefijo ‘is’.

```
# correcto
esVisible
isVisible

# incorrecto
Visible
```

Existen identificadores de funciones y variables de R que utilizan un punto para separar las palabras de un nombre. Opcionalmente, se puede aplicar este criterio.

```
# correcto
read.value
write.value
```

### 5.1.3. Funciones

Los nombres de las funciones se escriben aplicando el estilo ‘UpperCamelCase’. Siempre que sea posible, se deben utilizar verbos para los nombres de las funciones. Para métodos que devuelven un valor, se recomienda utilizar el prefijo ‘get’.

```
# correcto
CalcularCovarianza <- function(x, y)
getId <- function(x)

# incorrecto
calcular_covarianza <- function(x, y)
```

### 5.1.4. Clases

Los nombres de las clases se deben escribir aplicando el estilo 'UpperCamelCase'.

```
# correcto
RegistroInfantil

# incorrecto
registro.infantil
```

## 5.2. Sintaxis

Las siguientes recomendaciones se refieren a la sintaxis de un programa y definen su estructura y forma.

### 5.2.1. Longitud de las líneas de código

El tamaño máximo de una línea de código debe ser de 80 caracteres.

### 5.2.2. Uso de los espacios en blanco

Utilice espacios en blanco para separar los operadores binarios '+', '-', '*', '/', '^' de los operandos de una expresión.

```
# correcto
a <- (b + c) * 2

# incorrecto
a <- (b+c)*2
```

Utilice un espacio en blanco delante de un paréntesis abierto, excepto en la llamada a una función.

```
# correcto
esNegativo(x)

# incorrecto
esNegativo (x)
```

Evite el uso de espacios alrededor del operador '=' en los argumentos de las funciones.

```
# correcto
CalcularCovarianza <- function(x, y, print=TRUE)

# incorrecto
CalcularCovarianza <- function(x, y, print = TRUE)
```

Utilice una o más líneas de código para facilitar la lectura de los argumentos de una función.

```
# correcto
CalcularCovarianza <- function(x=xdata,
                               y=ydata,
                               print=TRUE)

# incorrecto
CalcularCovarianza <- function(x     = xdata,
                               y     = ydata,
                               print = TRUE)
```

No utilice un espacio antes de una coma, pero siempre después de ella.

```
# correcto
total <- sum(x[, 1])
total <- sum(x[1, ])

# incorrecto
total <- sum(x[,1])
total <- sum(x[1,])
```

### 5.2.3. Sangría del código

Utilice dos espacios como sangría de código. Evite el uso de tabuladores o de una combinación de espacios y tabuladores. Si una línea termina y se ha dejado un paréntesis abierto, la línea siguiente se debe alinear con el primer carácter dentro del paréntesis abierto.

```
# correcto
if (is.null(x))
  stop("¡El valor de x es nulo!")

# incorrecto
if (is.null(x))
stop("¡El valor de x es nulo!")
```

### 5.2.4. Uso de llaves

La llave que abre un bloque no debe escribirse en una línea nueva. Por el contrario, la llave que cierra un bloque debe escribirse en su propia línea.

```
# correcto
if (x >= 0) {
  y <- x * 10
  z <- x * 2
}

if (i == 0) {
  j <- 0
} else {
  j <- i * 2
}
```

```
# incorrecto
if (x >= 0)
{
  y <- x * 10
  z <- x * 2
}

if (i == 0) {
  j <- 0
}
else {
  j <- i * 2
}
```

### 5.2.5. El operador de asignación

Utilice siempre el operador '`<-`' para realizar una asignación. Evite en cualquier caso el uso del operador '`=`'.

```
# correcto
x <- 2

# incorrecto
x = 2
```

### 5.2.6. Uso de return(), invisible()

Las funciones deben utilizar una sola función `return` justo antes de la llave que cierra el bloque.

```
# correcto
esNegativo <- function(x){
  return (x < 0)
}

# incorrecto
esNegativo <- function(x){
  if (x < 0)
    return(TRUE)
  else
    return(FALSE)
}
```

## 5.3. Comentarios del código

Cada línea de comentario comienza por el carácter '#' seguido de un espacio. Se recomienda aplicar la misma sangría a los comentarios y al código.

## 5.4. Objetos y métodos

R utiliza dos tipos de clases distintos, las clases S3 y las clases S4. La definición de clases S3 es rápida y poco formal. Las clases S4, en cambio, son formales y rigurosas. Utilice las clases S3 o S4 en función de la complejidad del programa. Las clases S4 son necesarias cuando se diseña una estructura jerárquica de clases y subclases.

## 5.5. Manejo de excepciones

Todas las funciones deben manejar excepciones y utilizar la función `stop`.

```
CovarianzaC <- function(x, y) {
  n <- length(x)

  if (n <= 1)
    stop("¡El número de elementos del vector x debe ser mayor de 1!")
  else
    if (n != length(y))
      stop("¡Los vectores x, y deben tener el mismo númeo de elementos!")

      if (TRUE %in% is.na(x) || TRUE %in% is.na(y)) {
        stop("!Los vectores x, y no pueden tener valores NA!")
      }

  if (!is.loaded("ccov"))
    dyn.load("LibC.dll")

  out <- .C("ccov", x=as.double(x), y=as.double(y), n=as.integer(length(x)),
                    sxy=as.double(1))

  return(out$sxy)
}
```

## 5.6. Documentación de funciones

Se recomienda incluir una sección de comentarios en la línea siguiente a la declaración de una función. Los comentarios deben incluir la lista de los argumentos de la función y su valor de retorno. Los comentarios deben ser suficientemente descriptivos para que baste con leerlos para utilizar la función, sin necesidad de leer todo el código.

```
RegistroInfantil <- function(id, mes, estatura, peso, perimetro.craneal) {
  # método constructor de la clase RegistroInfantil, devuelve un objeto
  # con dos stributos: un identificador (id) y un data.frame (datos)

  x <- list(id=id, datos=data.frame(mes=mes, estatura=estatura,
            peso=peso, perimetro.craneal=perimetro.craneal))

  class(x) <- "RegistroInfantil"

  return(x)
}
```

# Capítulo 6. Desarrollo de paquetes en R

Un paquete es una extensión de R. El desarrollo de paquetes permite la libre distribución de software destinado a cubrir necesidades específicas de análisis de datos de diversas especialidades y contribuye a ampliar las capacidades del lenguaje.

Un paquete R normalmente almacena funciones y datos, encapsula el código fuente de las funciones y facilita al usuario su uso y la manipulación de los datos. Dependiendo de la funcionalidad de un paquete, el proceso de desarrollo puede ser complejo, pero una vez finalizado, facilita la distribución de software y datos aplicables a especialidades como la estadística o la bioinformática, entre otras. Además, los paquetes permiten hacer un uso más eficiente de la memoria, ya que R gestiona la carga y la descarga dinámica de las funciones almacenadas [11, 24].

Un paquete almacena código fuente, datos y documentación en el formato estándar de R. El código fuente de un paquete contiene texto y código de las funciones R almacenadas [5, 17, 18, 26]. Las versiones compiladas de los paquetes, solo se pueden utilizar en una plataforma determinada: Windows, Linux o Mac OS.

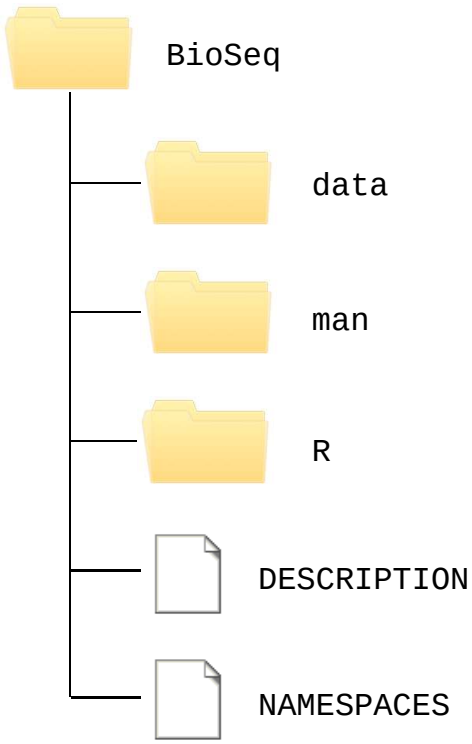
El desarrollo de un paquete requiere de un esfuerzo adicional al trabajo necesario para programar un conjunto de funciones. El código fuente de R no solo debe funcionar correctamente, sino que además debe cumplir con todos los requisitos de codificación y documentación que se exigen a los paquetes R. El proceso de verificación de un paquete es muy estricto. Se comprueba la estructura del paquete y el contenido de todos sus elementos, se verifica la correcta codificación de las clases y sus métodos, se analiza la coherencia entre el código fuente R y los archivos de documentación, se compara la estructura y los tipos de datos con la documentación de los archivos de datos almacenados, se verifica que los ejemplos de las funciones se ejecutan correctamente, se comprueban las dependencias con otros paquetes, etc.

## 6.1. Estructura de un paquete R

Un paquete de R se compone de los ficheros `DESCRIPTION`, `NAMESPACES` y de las carpetas `man`, `R`, `data`, `src`, `test`, `exec`. La carpeta `man` almacena la documentación de las funciones del paquete, `R` almacena el código delas funciones `R`, `data` almacena los ficheros de datos, `src` almacena los ficheros C, C++ y FORTRAN, `test` almacena los test de validación y `exec` es para misceláneos [13].

Para crear la carpeta de directorios de un paquete se ejecuta la función R `package.skeleton(name="package", code_files="source-file")`. Por ejemplo, para crear de forma automática el directorio del paquete y los ficheros de

documentación de `BioSeq`, se indica el nombre de la carpeta y el fichero que almacena el código R: `package.skeleton(name="BioSeq", code_files="BioSeq.R")`. El siguiente esquema muestra la estructura de carpetas del paquete de `BioSeq`.




## 6.2. El contenido de un paquete

A continuación se muestran ejemplos de los ficheros que componen el paquete `BioSeq`.

### 6.2.1. El fichero DESCRIPTION

El fichero `DESCRIPTION` almacena la información general sobre un paquete: el nombre del paquete, el título, la fecha de creación, la versión, el nombre de los autores y los responsables del mantenimiento, la compatibilidad con las versiones de R, una breve descripción y la licencia de uso.

```
Package: BioSeq
Type: Package
Title: Sequence Alignment
Date: 2013-06-06
Version: 1.0
Author: Victoria Lopez, Beatriz Gonzalez et al.
Maintainer: Beatriz Gonzalez<beatrizg@estad.ucm.es>
Depends: R (>= 2.15.1), seqinr
Description: Sequence Alignment and Database Management for Bioinformatics
License: GPL (>= 2)
```

El apartado `Depends: (>= 2.15.1), seqinr` especifica la versión de R y la lista de paquetes requeridos.

### 6.2.2. El fichero NAMESPACES

El fichero `NAMESPACES` se utiliza para indicar las funciones públicas del paquete.

```
exportPattern("^[[:alpha:]]+")
```

### 6.2.3. Los ficheros de documentación

Los ficheros de documentación describen el paquete, las funciones R y las bases de datos. El archivo de documentación de un paquete detalla el nombre, su alias, título y una breve descripción, la versión del paquete, la fecha de publicación, el tipo de licencia de uso, el nombre de los autores y de la persona responsable del mantenimiento, referencias sobre el paquete y ejemplos de uso. El siguiente ejemplo muestra el fichero de documentación del paquete `BioSeq`.

```
\name{BioSeq-package}
\alias{BioSeq-package}
\alias{BioSeq}
\docType{package}
\title{Sequence Alignment and Biological Database Management}
\description{
Local and global sequence alignment using Smith-Waterman and
Needelman-Wunsch algorithms}
\details{
\tabular{ll}{
Package: \tab BioSeq\cr
Type: \tab Package\cr
Version: \tab 1.0\cr
Date: \tab 2013-06-06\cr
License: \tab GPL (>= 2)\cr
}
This package implements two sequence alignment algorithms}
\author{Victoria Lopez, Beatriz Gonzalez et al.
Maintainer: Beatriz Gonzalez<beatrizg@estad.ucm.es>
}
\references{
Local ang global sequence alignment}
\keyword{ package }
\examples{
x <- SmithWaterman(
     c("A","T","C","G","T”,"A","T","T","C","G","G","T","C","A","A","C","T"),
     c("G","T","A","T","C","A","A","T","T","G","C","T","A","C","C"), -5,
     c("A","G","C","T"),
     c(10,-1,-3,-4,-1,7,-5,-3,-3,-5,9,0,-4,-3,0,8))
print(x)
plot(x)
y <- NeedlemanWunsch(
     c("A","T","C","G","T","A","T","T","C","G","G","T","C","A","A","C","T"),
     c("G","T","A","T","C","A","A","T","T","G","C","T","A","C","C"), -5,
     c("A","G","C","T"),
     c(10,-1,-3,-4,-1,7,-5,-3,-3,-5,9,0,-4,-3,0,8))
print(y)
plot(y)
}
```

El archivo de documentación de una función detalla el nombre de la función, su alias, título y una breve descripción, la declaración formal de la función y sus argumentos, el valor de retorno, ejemplos de uso y palabras clave de búsqueda. El siguiente ejemplo es

un fichero de documentación de la función `SmithWaterman` para alineamiento local de secuencias.

```
\name{SmithWaterman}
\alias{SmithWaterman}
\title{Smith-Waterman Local Sequence Alignment}
\description{
Local Sequence Alignment Using Smith-Waterman Algorithm
}
\usage{
SmithWaterman(seq1, seq2, gap, MSHeader, MSData)
}
\arguments{
\item{seq1}{vector of sequence 1}
\item{seq2}{vector of sequence 2}
\item{gap}{integer indicating the penalty value for gap}
\item{MSHeader}{vector of the score matrix header}
\item{MSData}{vector of score matrix data}
}
\value{
This function returns an object of class AlignedSequence, which contains
attributes seq1, seq2, score.mat, align.mat, opt.align1, opt.align2, score,
and gaps
}
\examples{
x                                                                        <-
SmithWaterman(c("A","T","C","G","T","A","T","T","C","G","G","T","C","A","A",
"C","T"),

c("G","T","A","T","C","A","A","T","T","G","C","T","A","C","C"),
                  -5,
                  c("A","G","C","T"),
                  c(10,-1,-3,-4,-1,7,-5,-3,-3,-5,9,0,-4,-3,0,8))
print(x)
plot(x)
}
\keyword{Bioinformatics, Local Sequence Aligment, Smith-Waterman}
```

El archivo de documentación de un archivo de datos detalla el nombre del conjunto de datos, su alias, título y una breve descripción, la descripción de cada campo de datos, la fuente de los datos, las referencias sobre los responsables de la elaboración de los datos, ejemplos de uso y palabras clave.

El siguiente ejemplo es un fichero de documentación del archivo de datos `Diesel`. Este archivo se compone de 227 repostajes de combustible recogidos entre 2003 y 2012. Cada registro de la tabla almacena la fecha, los litros repostados, el importe en euros y los kilómetros totales. El campo fecha almacena valores de tipo `Date` y los campos litros, euros y kilómetros son vectores numéricos.

```
\name{Diesel}
\alias{Diesel}
\docType{data}
\title{
Diesel dataset
}
\description{
Diesel dataset having liters, euros and km from 2003 to 2012
}
\usage{data(Diesel)}
\format{
  A data frame with 227 observations on the following variables
  \describe{
    \item{\code{Fecha}}{a Date}
    \item{\code{Litros}}{a numeric vector}
    \item{\code{Euros}}{a numeric vector}
    \item{\code{Km.total}}{a numeric vector}
  }
}
\source{
Dataset based on observations
}
\references{
Beatriz Gonzalez Perez, Mathematics Faculty
Universidad Complutense de Madrid (2013)
}
\examples{
data(Diesel)
}
\keyword{datasets}
```

### 6.2.4. Los ficheros de código R

El fichero `BioSeq.R` almacena el código R del paquete. Se compone de las funciones `Smitwhwaterman` y `Needlemanwunsch` para alineamiento de secuencias y de la declaración de la clase `AlignedSequence` para almacenar y manipular los resultados de las funciones de alineamiento. El código se incluye en el anexo A.

### 6.2.5. Los ficheros de datos

Los ficheros de datos son archivos de tipo `RDA`. Para exportar una tabla de datos con formato `RDA` es necesario cargar los datos en R para después almacenarlos con el formato de un `dataset` [16, 23]. Por ejemplo, el archivo de datos `Diesel` se obtiene de un fichero `CSV` y se exporta a formato `RDA`.

```
> setwd("c:/Mis documentos de trabajo/R Packages/Datasets")
> Diesel <- read.csv("Diesel.csv", header=TRUE, sep=";")
> save(Diesel, file="Diesel.rda")
```

Para crear el fichero de documentación del archivo de datos:

```
> prompt(Diesel)
Created file named ‘Diesel.Rd’.
```

## 6.3. El proceso de verificación y compilación de un paquete

Una vez que se han editado los ficheros de documentación del paquete, se puede realizar el proceso de verificación y compilación.

Si se trabaja en Windows, es necesario instalar RTools[17] y MiKTeX[18]. RTools incluye las aplicaciones necesarias para verificar y compilar un paquete, MiKTeX es una herramienta para generar los archivos de ayuda. Una vez instalado el software, se debe modificar la variable `PATH` para incluir los directorios de las siguientes aplicaciones y recursos:

- Los ficheros ejecutables, las librerías y el compilador de Rtools

  ```
  c:\Rtools\bin

  c:\Rtools\gcc-4.6.3\bin

  c:\Rtools\MinGW\bin
  ```

- Los ficheros ejecutables, las librerías y el API de R

  ```
  c:\Archivos de programa\R\R-3.0.1\bin

  c:\Archivos de programa\R\R-3.0.1\bin\i386

  c:\Archivos de programa\R\R-3.0.1\include
  ```

- Los archivos ejecutables de MiKTeX

  ```
  c:\Archivos de programa\MiKTeX 2.9\miktex\bin
  ```

El proceso de verificación del paquete se realiza en la consola de comandos del sistema operativo. Inicie una sesión de la consola y cambie el directorio actual por el directorio donde se almacena la carpeta del paquete. Si el proceso de verificación no detecta incidencias se generan los ficheros '`tarball`'[19] y `ZIP` del paquete. En caso contrario, el compilador indica el error que se ha producido. Para cualquier duda sobre el proceso de verificación de paquetes, consulte el documento "Writing R Extensions".

[17] http://cran.r-project.org/bin/windows/Rtools/
[18] http://miktex.org/
[19] Los ficheros 'tarball' y ZIP son archivos comprimidos.

La variable PATH:

```
PATH=c:\Rtools\bin;c:\Rtools\gcc-4.6.3\bin;c:\Rtools\MinGW\bin;
c:\Archivos de programa\MiKTeX 2.9\miktex\bin;
c:\Archivos de programa\R\R-3.0.1\include;
c:\Archivos de programa\R\R-3.0.1\bin;
c:\Archivos de programa\R\R-3.0.1\bin\i386;
c:\WINDOWS\system32;c:\WINDOWS;c:\WINDOWS\System32\Wbem;
c:\Archivos de programa\Archivos comunes\Adobe\AGL;
```

El comando `R CMD` que permite realizar las siguientes operaciones:

```
INSTALL    Instala un paquete
REMOVE     Elimina un paquete
SHLIB      Comila una librería de carga dinámica (DLL)
BATCH      Ejecuta R en modo batch
build      Genera el fichero 'tarball' de un paquete
check      Verifica un paquete
Rd2pdf     Convierte un fichero Rd a PDF
Rd2txt     Convierte un fichero Rd a texto
config     Muestra las opciones de configuración de R
textify    Procesa un fichero LaTeX
```

Para verificar un paquete se ejecuta `R CMD check`, para generar el fichero '`tarball`' `R CMD build` y para generar el fichero `ZIP`, se utiliza `R CMD INSTALL –-build`. Al final de todos estos comandos se debe indicar el nombre del paquete:

```
R CMD check BioSeq
R CMD build BioSeq
R CMD INSTALL –-build BioSeq
```

El comando `R CMD check BioSeq` ejecuta el proceso de verificación del paquete y muestra los resultados en la pantalla.

```
* using log directory 'C:/R Packages/BioSeq.Rcheck'
* using R version 3.0.1 (2013-05-16)
* using platform: i386-w64-mingw32 (32-bit)
* using session charset: ISO8859-1
* checking for file 'BioSeq/DESCRIPTION' ... OK
* checking extension type ... Package
* this is package 'BioSeq' version '1.0'
* checking package namespace information ... OK
* checking package dependencies ... OK
* checking if this is a source package ... OK
* checking if there is a namespace ... OK
* checking for executable files ... OK
* checking for hidden files and directories ... OK
* checking for portable file names ... OK
* checking whether package 'BioSeq' can be installed ... OK
* checking installed package size ... OK
* checking package directory ... OK
* checking DESCRIPTION meta-information ... OK
* checking top-level files ... OK
* checking for left-over files ... OK
* checking index information ... OK
* checking package subdirectories ... OK
* checking R files for non-ASCII characters ... OK
* checking R files for syntax errors ... OK
* checking whether the package can be loaded ... OK
* checking whether the package can be loaded with stated dependencies ... OK
```

```
* checking whether the package can be unloaded cleanly ... OK
* checking whether the namespace can be loaded with dependencies ... OK
* checking whether the namespace can be unloaded cleanly ... OK
* checking loading without being on the library search path ... OK
* checking for unstated dependencies in R code ... OK
* checking S3 generic/method consistency ... OK
* checking replacement functions ... OK
* checking foreign function calls ... OK
* checking R code for possible problems ... OK
* checking Rd files ... OK
* checking Rd metadata ... OK
* checking Rd cross-references ... OK
* checking for missing documentation entries ... OK
* checking for code/documentation mismatches ... OK
* checking Rd \usage sections ... OK
* checking Rd contents ... OK
* checking for unstated dependencies in examples ... OK
* checking contents of 'data' directory ... OK
* checking data for non-ASCII characters ... OK
* checking data for ASCII and uncompressed saves ... OK
* checking examples ... OK
* checking PDF version of manual ... OK
```

El comando `R CMD build BioSeq` genera el fichero '`tarball`' del paquete.

```
* checking for file 'BioSeq/DESCRIPTION' ... OK
* preparing 'BioSeq':
* checking DESCRIPTION meta-information ... OK
* checking for LF line-endings in source and make files
* checking for empty or unneeded directories
* looking to see if a 'data/datalist' file should be added
* building 'BioSeq_1.0.tar.gz'
```

Por último, el comando `R CMD INSTALL –-build BioSeq` genera el fichero `ZIP` del paquete.

```
* installing to library 'C:/Mis documentos/R/win-library/3.0'
* installing *source* package 'Bioseq'
** R
** data
** preparing package for lazy loading
** help
*** installing help indices
** building package indices
** testing if installed package can be loaded
* MD5 sums
package installation of 'BioSeq' as BioSeq_1.0.zip
* DONE <Bioseq>
```

Una vez creado el fichero `ZIP`, el paquete se puede instalar ejecutando el comando `utils:::menuInstallLocal()` de R.

```
> utils:::menuInstallLocal()
package ‘BioSeq’ successfully unpacked and MD5 sums checked
> library(BioSeq)
Loading required package: seqinr
> search()
 [1] ".GlobalEnv"        "package:BioSeq"    "package:seqinr"
 [4] "package:stats"     "package:graphics"  "package:grDevices"
 [7] "package:utils"     "package:datasets"  "package:methods"
[10] "Autoloads"         "package:base"
```

# Capítulo 7. La librería BioSeq para bioinformática

En el ámbito de la investigación en bioinformática se plantean problemas como la comparación y el alineamiento de secuencias, la comparación de estructuras de proteínas, el análisis de secuencias de nucleótidos y aminoácidos, el mapeo de cromosomas, la interpretación de expresiones genéticas, la construcción de árboles filogenéticos o el análisis de perfiles de expresión génica en microarrays. De todos estos problemas, el análisis de secuencias y la comparación de estructuras de proteínas son de especial relevancia en las investigaciones de esta disciplina [10, 13].

El ámbito de aplicación de la librería `BioSeq`[20] es fundamentalmente académico. Su objetivo es aportar técnicas y herramientas orientadas a resolver problemas específicos de la bioinformática, como la comparación y el alineamiento de secuencias o la gestión de bases de datos biológicas. La primera versión de esta librería implementa dos algoritmos de alineamiento de secuencias. El algoritmo de Smith-Waterman para alineamiento local y el de Needleman-Wunsch para alineamiento global. En el futuro, se espera que los alumnos de Bioinformática del Máster en Investigación en Informática utilicen esta librería e implementen nuevos algoritmos para ampliar su funcionalidad en futuras versiones. `BioSeq` ofrece tres bases de datos desarrolladas por la Universidad Complutense de Madrid, denominadas `Diesel`, `Gasolina95` y `Cabras`. Las dos primeras almacenan lecturas de consumos de combustible y la última recoge datos morfológicos de 531 cabras. La base de datos `Diesel` almacena consumos de combustible desde el año 2003 hasta 2012, incluye los litros repostados, el coste en euros y el kilometraje realizado. `Gasolina95` almacena consumos de combustible desde el año 2008 hasta 2012, incluye los litros repostados, el coste en euros y el kilometraje realizado. Las bases de datos `Diesel` y `Gasolina95` fueron elaboradas por Beatriz González Pérez, del Departamento Estadística de la Facultad de Matemáticas. La base de datos `Cabras` almacena datos morfológicos de un rebaño de cabras compuesto por 531 machos y hembras con edades que van desde los 2 hasta los 4 años, clasificados en las categorías Andosco, Trasandosco y Cerrado. La categoría Andosco se refiere a cabras con edad mayor o igual a dos y menor de 3 años, Trasandosco es para cabras con edad mayor o igual a 3 y menor de 4 años y Cerrado para cabras mayores o iguales a 4 años. Estos datos se tomaron en el año 1997 de una población de cabras del Guadarrama compuesta por 7236 hembras y 204 machos. Esta información fue

[20] La librería BioSeq 1.0 está disponible en el apartado Bioinformática y bioestadística de la página: http://www.tecnologiaucm.es

elaborada por Jesús de la Fuente Vázquez, del Departamento de Producción Animal de la Facultad de Veterinaria [10].

## 7.1. Funciones de alineamiento de secuencias

A continuación se describen las funciones de alineamiento local y alineamiento global de secuencias de la librería.

### 7.1.1. SmithWaterman

Descripción y ejemplo de uso de la función SmithWaterman.

Nombre y declaración

```
SmithWaterman(seq1, seq2, gap, MSHeader, MSData)
```

Argumentos

```
seq1: vector de la primera secuencia
seq2: vector de la segunda secuencia
gap: penalización por gap
MSHeader:vector de la cabecera de la matriz de puntuación
MSData: vector de los valores de la matriz de puntuación, dados por filas
```

Ejemplo de uso

```
> x <- SmithWaterman(
     c("A","T","C","G","T","A","T","T","C","G","G","T","C","A","A","C","T"),
     c("G","T","A","T","C","A","A","T","T","G","C","T","A","C","C"),
     -5,
     c("A","G","C","T"),
     c(10,-1,-3,-4,-1,7,-5,-3,-3,-5,9,0,-4,-3,0,8))
```

Resultados

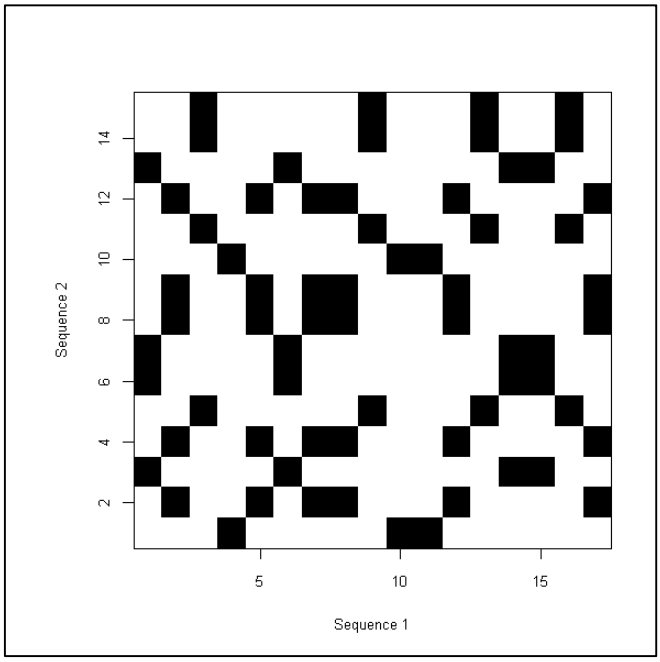
```
> print(x)
Sequence 1
A T C G T A T T C G G T C A A C T

Sequence 2
G T A T C A A T T G C T A C C

Score Matrix
   A G C T
A 10 0 0 0
G  0 7 0 0
C  0 0 9 0
T  0 0 0 8




Alignment Matrix
```

```
     A  T  C  G  T  A  T  T  C  G  G  T  C  A  A  C  T
  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
G 0  0  0  0  7  2  0  0  0  0  7  7  2  0  0  0  0  0
T 0  0  8  3  2 15 10  8  8  3  2  7 15 10  5  0  0  8
A 0 10  5  8  3 10 25 20 15 10  5  2 10 15 20 15 10  5
T 0  5 18 13  8 11 20 33 28 23 18 13 10 10 15 20 15 18
C 0  0 13 27 22 17 15 28 33 37 32 27 22 19 14 15 29 24
A 0 10  8 22 27 22 27 23 28 33 37 32 27 22 29 24 24 29
A 0 10 10 17 22 27 32 27 23 28 33 37 32 27 32 39 34 29
T 0  5 18 13 17 30 27 40 35 30 28 33 45 40 35 34 39 42
T 0  0 13 18 13 25 30 35 48 43 38 33 41 45 40 35 34 47
G 0  0  8 13 25 20 25 30 43 48 50 45 40 41 45 40 35 42
C 0  0  3 17 20 25 20 25 38 52 48 50 45 49 44 45 49 44
T 0  0  8 12 17 28 25 28 33 47 52 48 58 53 49 44 45 57
A 0 10  5  8 12 23 38 33 28 42 47 52 53 58 63 59 54 52
C 0  5 10 14  9 18 33 38 33 37 42 47 52 62 58 63 68 63
C 0  0  5 19 14 13 28 33 38 42 37 42 47 61 62 58 72 68


Optimal Alignment of Sequence 1
A T C G T A T T C G G T C A A C

Optimal Alignment of Sequence 2
A T C - A A T T - G C T - A C C

Optimal Alignment Score
72

Gaps
3
```

## Gráficos

Los valores de ambos ejes representan la posición de los elementos de cada una de las secuencias. Los valores de la secuencia del eje X que son idénticos a los de la secuencia del eje Y, se marcan en negro en esa posición del gráfico.

```
> plot(x)
```



### 7.1.2. NeedlemanWunsch

Descripción y ejemplo de uso de la función NeedelmanWunsch.

Nombre y declaración

```
NeedlemanWunsch(seq1, seq2, gap, MSHeader, MSData)
```

Argumentos

```
seq1: vector de la primera secuencia
seq2: vector de la segunda secuencia
gap: penalización por gap
MSHeader:vector de la cabecera de la matriz de puntuación
MSData: vector de los valores de la matriz de puntuación, dados por filas
```

Ejemplo de uso

```
x <- NeedlemanWunsch(c("A","C","A","C","A","C","T","A"),
                     c("A","G","C","A","C","A","C","A"),
                     -5,
                     c("A","G","C","T"),
                     c(10,-1,-3,-4,-1,7,-5,-3,-3,-5,9,0,-4,-3,0,8))
```

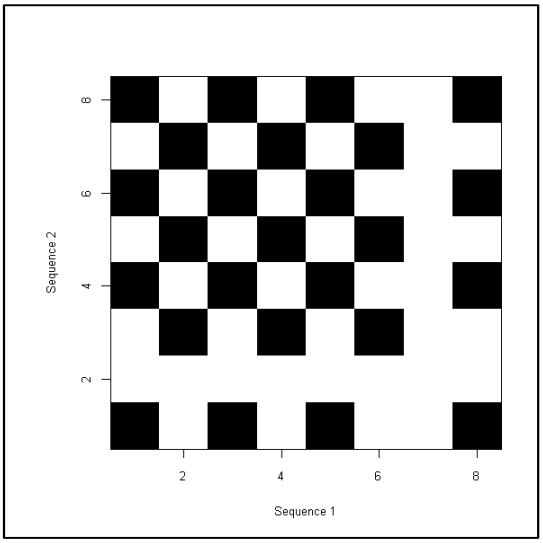
Resultados

```
> print(x)
Sequence 1
A C A C A C T A

Sequence 2
A G C A C A C A

Score Matrix
   A  G  C  T
A 10 -1 -3 -4
G -1  7 -5 -3
C -3 -5  9  0
T -4 -3  0  8

Alignment Matrix
        A   C   A   C   A   C   T   A
    0  -5 -10 -15 -20 -25 -30 -35 -40
A  -5  10   5   0  -5 -10 -15 -20 -25
G -10   5   5   4  -1  -6 -11 -16 -21
C -15   0  14   9  13   8   3  -2  -7
A -20  -5   9  24  19  23  18  13   8
C -25 -10   4  19  33  28  32  27  22
A -30 -15  -1  14  28  43  38  33  37
C -35 -20  -6   9  23  38  52  47  42
A -40 -25 -11   4  18  33  47  48  57

Optimal Alignment of Sequence 1
A - C A C A C T A

Optimal Alignment of Sequence 2
A G C A C A C - A



Optimal Alignment Score
```

```
57

Gaps
2
```

Gráficos

```
Los valores de ambos ejes representan la posición de los elementos de cada
una de las secuencias. Los valores de la secuencia del eje X que son
idénticos a los de la secuencia del eje Y, se marcan en negro en esa
posición del gráfico.

> plot(x)
```




Además de estas funciones de alineamiento de secuencias, se define la clase S3 `AlignedSequence` y sus métodos `print` y `plot`. El código R de todas las funciones se incluye en el apartado 7.3.

## 7.2. Bases de datos

La base de datos `Diesel` almacena 227 registros de repostajes realizados entre 2003 y 2012.

| Campo | Descripción |
|---|---|
| Fecha | Fecha del repostaje |
| Litros | Litros repostados |
| Euros | Importe en euros |
| Km.total | Kilómetros totales recorridos |

La tabla de datos `Gasolina95` almacena 109 registros de repostajes realizados entre 2008 y 2012.

| Campo | Descripción |
|---|---|
| Fecha | Fecha del repostaje |
| Litros | Litros repostados |
| Euros | Importe en euros |
| Km.total | Kilómetros totales recorridos |

La tabla de datos `Cabras` almacena 531 registros de datos morfológicos de un rebaño de cabras con edades entre los 2 y los 4 años.

| Campo | Descripción |
|---|---|
| Sexo | H: Hembra, M: Macho |
| Edad | A: Andosco, mayor o igual a 2 y mejor de 3 años<br>T: Trasandosco, mayor o igual a 3 y menor de 4 años<br>C: Cerrado, mayor o igual a 4 años |
| Altura.cruz | Altura de la cruz |
| Altura.dorso | Altura del dorso |
| Altura.grupa | Altura de la grupa |
| Altura.hueco | Altura del hueco |
| Diametro longitudinal | Diámetro longitudinal |
| Diametro.dorso | Diámetro del dorso |
| Diametro bicostal | Diámetro bicostal |
| Longitud.cabeza | Longitud de la cabeza |
| Ancho.cabeza | Anncho de la cabeza |
| Ancho.anterior grupa | Ancho anterior de la grupa |
| Ancho.posterior grupa | Ancho posterior de la grupa |
| Longitud.grupa | Longitud de la grupa |
| Ancho.cana | Ancho de la caña |
| Longitud.cuerno | Longitud del cuerno |
| Longitud.oreja | Longitud de la oreja |
| Perimetro toracico | Perímetro torácico |
| Perimetro.cana | Perímetro de la caña |
| Perimetro corvejon | Perímetro del corvejón |
| Peso | Peso |

## 7.3. El código R

A continuación se incluye el código R de la librería `BioSeq` versión 1.0[21].

```
#------------------------------------------------------------------------#
# BioSeq 1.0 package                                                     #
#------------------------------------------------------------------------#

# AlignedSequence Class for Results: constructor, print and # plot methods

AlignedSequence <- function(seq1, seq2, score.mat, align.mat,
                            opt.align1, opt.align2, score, gaps) {
  obj <- list(seq1=seq1, seq2=seq2, score.mat=score.mat,
              align.mat=align.mat, opt.align1=opt.align1,
              opt.align2=opt.align2, score=score, gaps=gaps)

  class(obj) <- "AlignedSequence"

  return(obj)
}

print.AlignedSequence <- function(x, ...) {
  cat("Sequence 1 \n")
  cat(x$seq1, "\n")
  cat("\nSequence 2 \n")
  cat(x$seq2, "\n")
  cat("\nScore Matrix \n")
  print(x$score.mat)
  cat("\nAlignment Matrix \n")
  print(x$align.mat)
  cat("\nOptimal Alignment of Sequence 1 \n")
  cat(x$opt.align1, "\n")
  cat("\nOptimal Alignment of Sequence 2 \n")
  cat(x$opt.align2, "\n")
  cat("\nOptimal Alignment Score \n")
  cat(x$score, "\n")
  cat("\nGaps \n")
  cat(x$gaps, "\n")
}
```

[21] La librería BioSeq 1.0 se ha desarrollada en colaboración con otros alumnos del Máster en Investigación en Informática. Los algoritmos SmithWaterman y NeedlemanWunsch han sido desarrollados por Óscar Sánchez.

```
plot.AlignedSequence <- function(x, y, ...) {
  if (is.element("seqinr", installed.packages()[,1])) {
    if (!("package:seqinr" %in% search())) library ("seqinr")
      seq1 <- x$seq1
      seq2 <- x$seq2

      par(oma=c(0, 2, 0, 0), ps=10)

      dotPlot(seq1=seq1, seq2=seq2, xlab="Sequence 1", ylab="Sequence 2")
  } else {
      stop("Library 'sequinr' must be installed to plot!")
  }
}



# Local Sequence Alignment using SmithWaterman Algorithm

SmithWaterman <- function(seq1, seq2, gap, MSHeader, MSData){
  #Contador de huecos.

  ContGaps <- 0

  #Longitud fila y columna de la matriz de sustitución.

  MSRowCol <- length(MSHeader)

  #Valores de una matriz cuadrada que hará la función de matriz de puntaje.

  NombresF <- NombresC <- MSHeader

  MatrizPuntaje <- matrix(MSData,MSRowCol,MSRowCol,byrow=TRUE,
                          dimnames=list(NombresC,NombresF))

  #Convertimos los valores negativos de la matriz de puntaje a 0.

  for (i in 1:length(NombresF)){
    for (j in 1:length(NombresC)){
      if(MatrizPuntaje[i,j] < 0){
        MatrizPuntaje[i,j] <- 0
      }
    }
  }

  #Matriz que muestra los resultados parciales de cada posible alineamiento.

  S1 <- c("",seq1)
  S2 <- c("",seq2)

  LengthS1 <- length(S1)
  LengthS2 <- length(S2)

  MatrizAlin <- matrix(data=NA, nrow=LengthS2, ncol=LengthS1, byrow=FALSE,
                       dimnames = list(S2,S1))

  for (i in 1:LengthS2){
    if(i==1) {
      MatrizAlin[i,1] <- 0
    } else{
      MatrizAlin[i,1] <- gap*i-gap
      if(MatrizAlin[i,1] < 0) {
        MatrizAlin[i,1] <- 0
      }
    }
  }
```

```
for (j in 1:LengthS1){
  if(j==1) {
    MatrizAlin[1,j] <- 0
  }else{
    MatrizAlin[1,j] <- gap*j-gap
    if(MatrizAlin[1,j] < 0) {
      MatrizAlin[1,j] <- 0
    }
  }
}

for (i in 2:LengthS2){
  for (j in 2:LengthS1){
    Elem1 <- S1[j]
    Elem2 <- S2[i]

    for (k in 1:length(NombresF)){
      if(Elem1==NombresF[k]){
        PosF <- k
      }
    }

    for (l in 1:length(NombresC)){
      if(Elem2==NombresC[l]){
        PosC <- l
      }
    }

    Diag <- MatrizAlin[i-1,j-1] + MatrizPuntaje[PosF,PosC]
    Arriba <- MatrizAlin[i-1,j] + gap
    Izq <- MatrizAlin[i,j-1] + gap
    MatrizAlin[i,j] <- max(Diag,Arriba,Izq)
  }
}

#Hallamos el alineamiento óptimo de las secuencias retrocediendo en la
#matriz de alineamiento.

AlineamA <- character()
AlineamB <- character()

MaxVal <- 0

for (i in 2:LengthS2){
  for (j in 2:LengthS1){
    if(MaxVal < MatrizAlin[i,j]){
      MaxVal <- MatrizAlin[i,j]
      MaxVali <- i
      MaxValj <- j
    }
  }
}

i <- MaxVali
j <- MaxValj

while((i > 1 & j > 1) | (MatrizAlin[i,j] > 0)){
  Punt <- MatrizAlin[i,j]
  Diag <- MatrizAlin[i-1,j-1]
  Arriba <- MatrizAlin[i-1,j]
  Izq <- MatrizAlin[i,j-1]

  Elem1 <- S1[j]
  Elem2 <- S2[i]
```

```
    for (k in 1:length(NombresF)){
      if(Elem1==NombresF[k]){
        PosF <- k
      }
    }

    for (l in 1:length(NombresC)){
      if(Elem2==NombresC[l]){
        PosC <- l
      }
    }

    if(Punt == Diag + MatrizPuntaje[PosF,PosC]) {
      AlineamA <- c(S1[j],AlineamA)
      AlineamB <- c(S2[i],AlineamB)
      i <- i-1
      j <- j-1
    }else if(Punt == Arriba + gap){
      AlineamA <- c("-",AlineamA)
      AlineamB <- c(S2[i],AlineamB)
      i <- i-1
      ContGaps <- ContGaps + 1
    }else if(Punt == Izq + gap){
      AlineamA <- c(S1[j],AlineamA)
      AlineamB <- c("-",AlineamB)
      j <- j-1
      ContGaps <- ContGaps + 1
    }
  }

  while((i > 1) & (MatrizAlin[i,j] > 0)) {
    AlineamA <- c("-",AlineamA)
    AlineamB <- c(S2[i],AlineamB)
    i <- i-1
    ContGaps <- ContGaps + 1
  }

  while((j > 1) & (MatrizAlin[i,j] > 0)) {
    AlineamA <- c(S1[j],AlineamA)
    AlineamB <- c("-",AlineamB)
    j <- j-1
    ContGaps <- ContGaps + 1
  }

  return(AlignedSequence(seq1=seq1,
                         seq2=seq2,
                         score.mat=MatrizPuntaje,
                         align.mat=MatrizAlin,
                         opt.align1=AlineamA,
                         opt.align2=AlineamB,
                         score=MatrizAlin[MaxVali,MaxValj],
                         gaps=ContGaps))
}



# Global Sequence Alignment using NeedlemanWunch Algoritm

NeedlemanWunsch<-function(seq1, seq2, gap, MSHeader,MSData) {
  #Contador de huecos.

  ContGaps <- 0

  #Longitud fila y columna de la matriz de sustitución.

  MSRowCol <- length(MSHeader)

  #Valores de una matriz cuadrada que hará la función de matriz de puntaje.
```

```
  NombresF <- NombresC <- MSHeader

  MatrizPuntaje <- matrix(MSData,MSRowCol,MSRowCol,byrow=TRUE,
                          dimnames=list(NombresC,NombresF))

#Matriz que muestra los resultados parciales de cada posible alineamiento.

  S1 <- c("",seq1)
  S2 <- c("",seq2)

  LengthS1 <- length(S1)
  LengthS2 <- length(S2)

  MatrizAlin <- matrix(data=NA, nrow=LengthS2, ncol=LengthS1, byrow=FALSE,
                       dimnames = list(S2,S1))

  for (i in 1:LengthS2){
    if(i==1) {
      MatrizAlin[i,1] <- 0
    }else{
      MatrizAlin[i,1] <- gap*i-gap
    }
  }

  for (j in 1:LengthS1){
    if(j==1) {
      MatrizAlin[1,j] <- 0
    }else{
      MatrizAlin[1,j] <- gap*j-gap
    }
  }

  for (i in 2:LengthS2){
    for (j in 2:LengthS1){
      Elem1 <- S1[j]
      Elem2 <- S2[i]

      for (k in 1:length(NombresF)){
        if(Elem1==NombresF[k]){
          PosF <- k
        }
      }

      for (l in 1:length(NombresC)){
        if(Elem2==NombresC[l]){
          PosC <- l
        }
      }

      Diag <- MatrizAlin[i-1,j-1] + MatrizPuntaje[PosF,PosC]
      Arriba <- MatrizAlin[i-1,j] + gap
      Izq <- MatrizAlin[i,j-1] + gap
      MatrizAlin[i,j] <- max(Diag,Arriba,Izq)
    }
  }
#Hallamos el alineamiento óptimo de las secuencias retrocediendo en la
#matriz de alineamiento.

  AlineamA <- character()
  AlineamB <- character()

  i <- LengthS2
  j <- LengthS1
```

```
  while(i > 1 & j > 1){

    Punt <- MatrizAlin[i,j]
    Diag <- MatrizAlin[i-1,j-1]
    Arriba <- MatrizAlin[i-1,j]
    Izq <- MatrizAlin[i,j-1]

    Elem1 <- S1[j]
    Elem2 <- S2[i]

    for (k in 1:length(NombresF)){
      if(Elem1==NombresF[k]){
        PosF <- k
      }
    }

    for (l in 1:length(NombresC)){
      if(Elem2==NombresC[l]){
        PosC <- l
   }
    }

    if(Punt == Diag + MatrizPuntaje[PosF,PosC]) {
      AlineamA <- c(S1[j],AlineamA)
      AlineamB <- c(S2[i],AlineamB)
      i <- i-1
      j <- j-1
    }else if(Punt == Arriba + gap){
      AlineamA <- c("-",AlineamA)
      AlineamB <- c(S2[i],AlineamB)
      i <- i-1
      ContGaps <- ContGaps + 1
    }else if(Punt == Izq + gap){
      AlineamA <- c(S1[j],AlineamA)
      AlineamB <- c("-",AlineamB)
      j <- j-1
      ContGaps <- ContGaps + 1
    }
}

  while(i > 1) {
    AlineamA <- c("-",AlineamA)
    AlineamB <- c(S2[i],AlineamB)
    i <- i-1
    ContGaps <- ContGaps + 1
  }

  while(j > 1) {
    AlineamA <- c(S1[j],AlineamA)
    AlineamB <- c("-",AlineamB)
    j <- j-1
    ContGaps <- ContGaps + 1
  }

  return(AlignedSequence(seq1=seq1,
                         seq2=seq2,
                         score.mat=MatrizPuntaje,
                         align.mat=MatrizAlin,
                         opt.align1=AlineamA,
                         opt.align2=AlineamB,
                         score=MatrizAlin[LengthS2,LengthS1],
                         gaps=ContGaps))
}
```

# Capítulo 8. Conclusiones

El desarrollo de paquetes en R es complejo y normalmente requiere conocimientos avanzados de programación. Si el paquete debe trabajar con grandes cantidades de datos, entonces es necesario aprovechar al máximo las funciones nativas de R o desarrollar librerías C externas específicas que respondan a las necesidades de cálculo del paquete.

Sin duda, el proceso de verificación y compilación de un paquete es muy exigente y obliga a desarrollar el código R correctamente, aplicando técnicas avanzadas como la programación orientada a objetos y el uso de librerías externas C. Por todo esto, es necesario tener un nivel avanzado de conocimientos de las clases S3 y S4 de R para conocer sus características y sus limitaciones. Por otra parte, tener conocimientos de C es prácticamente imprescindible, dado el impacto que tiene en la eficiencia de una aplicación. Como se ha visto en las pruebas realizadas, las librerías externas C son muy eficientes cuando se trabaja con grandes cantidades de datos y órdenes de magnitud mayores o iguales a 1E5. En estas condiciones, la función nativa R que calcula la covarianza entre dos vectores ha sido entre un 23% y un 42% más lenta que la librería externa C. La función R programada con algoritmos iterativos y sin funciones nativas R ha sido entre un 4300% y un 4900% más lenta.

Además de aplicar técnicas avanzadas de programación, es fundamental utilizar una metodología de desarrollo de software para realizar una correcta gestión del proyecto, desde la fase inicial de análisis y diseño, hasta el desarrollo, las pruebas y la documentación, ya que esto condiciona la calidad del producto final. Por otra parte, es muy importante realizar un estudio previo de las necesidades del paquete para determinar si la funcionalidad que se desea desarrollar ya existe en otros paquetes disponibles en el CRAN.

Esta primera versión de la librería `BioSeq` es resultado del trabajo de un grupo de alumnos de Bioinformática del Máster. Su objetivo es aportar técnicas y herramientas orientadas a resolver problemas específicos de la bioinformática, como la comparación y el alineamiento de secuencias o la gestión de bases de datos biológicas. Considero que las técnicas de programación abordadas en este trabajo serán de utilidad para que otros alumnos implementen nuevos algoritmos para ampliar la funcionalidad de la librería en futuras versiones.

Por último, en relación con el trabajo futuro, creo que es importante ampliar al apartado de desarrollo de paquetes para plataformas Linux y Mac OS. Por otra parte, se podría profundizar en el desarrollo de funciones externas C utilizando el 'interface' `.Call` y analizar las ventajas que aporta con respecto al 'interface' `.C` utilizado en este trabajo.

Sería interesante realizar un análisis comparativo del rendimiento de ambos ‘interfaces’ para saber cuál de ellos es más eficiente y en qué casos se debe aplicar uno u otro.

# Referencias bibliográficas

[1] Bååth, R. The State of Naming Conventions in R. The R Journal Vol. 4/12. December 2012. pp. 74-75.

[2] Becker, R. A. (1994). A brief history of S. Cahier de recherche, AT&T Bell Laboratories.

[3] Bengtsson, H. (2009). R Coding Conventions.
https://docs.google.com/document/d/1esDVxyWvH8AsX-VJa-8oqWaHLs4stGlIbk8kLc5VlII/edit?pli=1
http://www1.maths.lth.se/help/R/R.oo/#3. Coding conventions

[4] Bioconductor Coding Style
http://www.bioconductor.org/developers/coding-style/

[5] Carmona, F. (2013). Creación de paquetes de R en Wndows (y Linux).

[6] Chambers, J. M. (2009). Developments in Class Inheritance and Method Selection.

[7] Chambers, J. M. (2006). How S4 Methods Work. Technical report.

[8] Chambers, J. M. (2003). S4 Classes in 15 pages, more or less. Tehnical report.

[9] Chambers, J. M. (2001) Classes and Methods in the S Language. Technical report.

[10] Cordero, J., Martínez, J. Sánchez, O., López, V. González, B. BioSeq: Una librería R para el análisis de secuencias de datos. Actas del Simposio Lógica Fuzzy y Soft Computing del Congreso Español de Informática, CEDI 2013, aceptado, pendiente de publicación.

[11] Emerson, J. R (2012). Packages and the C/C++ Interface.

[12] Genolini, C. (2008). A (Not So) Short Introduction to S4. Technical report.

[13] González-Pérez, B., López, V., Sampedro, J. (2012). Programming Global and Local Sequence Alignment by using R. Proceedings of ISKE.

[14] Google´s R Style Guide
http://google-styleguide.googlecode.com/svn/trunk/google-r-style.html

[15] Kernighan, B.W., Ritchie, D.M. (1988). The C Programming Language. Second Edition. Prentice Hall Software Series.

[16] Lundholm, M. (2010). Loading Data into R.

[17] Leisch, F. (2008). Creating R Packages: A Tutorial.

[18] Nunes, Matt. (2010). Creating and building R packages: course notes.

[19] Paradis, E. (2002). R for Beginners.

[20] Peng, R.D., Leeuw, J. (2002). An Introduction to the .C Interface to R.

[21] R Code Conventions (CellNOpt Documentation Center).
http://www.cellnopt.org/doc/cnodocs/R_code_layout.html

[22] R Development Core Team. R: A Language and Environment for Statistical Computing. Reference Index. Version 2.15.0 (2012-03-30).

[23] R Development Core Team. R Data Import/Export. Version 2.15.0 (2012-03-30)

[24] R Development Core Team. Writing R Extensions. Version 2.15.0 (2012-03-30)

[25] R Development Core Team. R Language Definition. Version 2.15.0 (2012-03-30) DRAFT.

[26] Rossi, P. (2006). Making R packages under windows: A tutorial. Tech. Rep.

[27] Scott, T. A. An Introduction to the Fundamentals & Functionality of the R Programming Language. Part I. An Overview.

[28] Scott, T. A. An Introduction to the Fundamentals & Functionality of the R Programming Language. Part II. The Nuts and Bolts.

[29] Venables, W. N., Smith, D. M. (2012). An Introduction to R. Notes on R: A programming Environment for Data Analysis and Graphics

[30] Wickman, H. R Style Guide.
http://stat405.had.co.nz/r-style.html